NASA Technical Memorandum 108995

Parallel Software Tools at Langley Research Center

Stuti Moitra, Geoffrey M. Tennille, Christopher D. Lakeotes, Donald P. Randall, Jarvis J. Arthur, Dana P. Hammond, and Gerald H. Mall

March 1993

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Table of Contents

Chapter 1
INTRODUCTION

1.1 High Performance Computing at NASA Langley 1-1

1.2 Evaluation of Parallel Software Tools 1-2

1.3 Future Direction 1-3

Chapter 2
Express - Basic Tools

2.1 Overview of Express 2-1

2.2 Access to Express 2-2

2.2.1 Express and NetExpress 2-2

2.3 Detailed Description of Express 2-3

2.3.1 CUBIX Input/Output 2-3
2.3.1.1 Advantages and Disadvantages of the CUBIX Programming Model 2-6

2.3.2 Host/Node Model 2-6
2.3.2.1 Advantages and Disadvantages of the Host/Node Model 2-6

2.3.3 Compiling CUBIX and Host/Node Programs 2-7

2.3.4 Executing Express CUBIX Programs 2-9
2.3.4.1 Sample CUBIX and Host/Node Programs 2-10

2.3.5 Express System Calls 2-16
2.3.5.1 System Initialization 2-16
2.3.5.2 Processor Allocation and Cube Control 2-17
2.3.5.3 Data Decomposition 2-17
2.3.5.4 Message - Passing 2-18
2.3.5.5 Global Operations 2-19
2.3.5.6 I/O 2-19
2.3.5.7 Utility Routines 2-20

2.4 Observations on Using Express 2-21

2.5 Known Bugs of Express 2-23

2.6 References for Express 2-23

Chapter 3
Express - Performance Analysis Tool (PM)

3.1 Overview of PM 3-1

3.2 Access to PM 3-1

3.3 Detailed Description of PM 3-1
3.3.1 Using the Program Execution Tool (xtool) 3-1
3.3.2 Using the Program Communication Tool (ctool) 3-2
3.3.3 Using the Program Events (etool) 3-2
3.3.4 Compiling and Executing a CUBIX PM Program 3-3

3.4 Observations on Using PM 3-4
3.4.1 Data Gathering Using Compiler Switches 3-4
3.4.2 Data Gathering Using the Programmatic Interface 3-4

3.5 Known Bugs of PM 3-5

3.6 References for PM 3-5

Chapter 4
Express - Graphics Tool (PLOTIX)

4.1 Overview of PLOTIX 4-1

4.2 Access to PLOTIX 4-1

4.3 Detailed Description of PLOTIX 4-2

4.4 Observations on Using PLOTIX 4-3

4.5 Known Bugs of PLOTIX 4-3

4.6 References for PLOTIX 4-3

Chapter 5
Interactive Parallel Debugger (IPD)

5.1 Overview of IPD 5-1

5.2 Access to IPD 5-1

5.2.1 Invoking IPD from Bluecrab 5-1
5.2.2 Node Program Specifications 5-2
5.2.3 Exclusive IPD Features 5-2

5.3 Detailed Description of IPD 5-2

5.3.1 Node Program Compilation 5-2

5.3.2 Other Preparation 5-3

5.3.3 Asynchronous Node Program Execution Monitoring 5-3

5.3.4 Some Diagnostic Approaches at Halt 5-4

5.4 Observations on Using IPD 5-4

5.4.1 Strengths 5-5

5.4.2 Weaknesses 5-5

5.5 Known Bugs of IPD 5-7

5.6 References for IPD 5-7

Chapter 6
MIMDizer

6.1 Overview of MIMDizer 6-1

6.2 Access to MIMDizer 6-2

6.3 Detailed Description of MIMDizer 6-2

6.4 Observations on Using MIMDizer 6-4

6.5 Known Bugs of MIMDizer 6-4

6.6 References for MIMDizer 6-4

Chapter 7
Performance Analysis Tool (PAT)

7.1 Overview of PAT 7-1

7.2 Access to PAT 7-1

7.3 Detailed Description of PAT 7-2

7.3.1 Profiling Methods 7-3
7.3.1.1 Profiling the Program Execution 7-3
7.3.1.2 Profiling the Program Communication 7-4
7.3.1.3 Profiling the Program Events 7-5

7.3.2 Analyzing Methods 7-6
7.3.2.1 Analyzing Execution Profile (xtool) 7-8
7.3.2.2 Analyzing Communication Profile (ctool) 7-9
7.3.2.3 Analyzing Event Profile (etool) 7-11

7.3.3 Compilation and Execution 7-12

7.3.4 System Calls 7-15

7.4 Observations on Using PAT 7-17

7.5 Known Bugs of PAT 7-17

7.6 References for PAT 7-17

Chapter 8
Parallel Virtual Machine (PVM)

8.1 Overview of PVM 8-1

8.1.1 OS and Platforms 8-2

8.2 Access to PVM 8-2

8.3 Detailed Description of PVM 8-3

8.3.1 Compiling with PVM 8-3

8.3.2 Running a PVM Program 8-5

8.3.3 Description of FORTRAN Subroutines in PVM 8-6

8.3.4 Global Subroutines 8-9

8.4 Observations on Using PVM 8-10

8.5 Known Bugs of PVM 8-11

8.6 References for PVM 8-11

ACRONYMS

ACD	Analysis and Computation Division
BLAS	Basic Linear Algebra Subroutines
CAB	Computer Applications Branch
CAS	Computational Aero-Sciences
CFS	Concurrent File System
CPU	Central Processing Unit
CSCC	Central Scientific Computing Complex
DNS	Domain Name Service
ESS	Earth and Space Sciences
FFT	Fast Fourier Transform
FTP	File Transfer Protocol
GFLOPS	Gigaflops (Giga[billion] FLoating point OPerations per Second)
GSFC	Goddard Space Flight Center
HPCCP	High Performance Computing and Communications Program
ICASE	Institute for Computer Applications in Science and Engineering
I/O	Input/Output
IPD	Interactive Parallel Debugger
LaRC	Langley Research Center
MFLOPS	Megaflops (Mega[million] FLoating Point OPerations per Second)
MIMD	Multiple-Instruction, Multiple-Data
MSS	Mass Storage Subsystem
NAS	Numerical Aerodynamic Simulation
NFS	Network File System
NQS	Network Queuing System
OCO	Operations Control Office
ORNL	Oak Ridge National Laboratory
PAT	Performance Analysis Tool
PVM	Parallel Virtual Machine
SNS	Supercomputing Network Subsystem
SPMD	Single Program Multiple Data
SRM	System Resource Manager
TCP/IP	Transmission Control Protocol/Internet Protocol
X,X11	X Window System

1 INTRODUCTION

1.1 High Performance Computing at NASA Langley

The Central Scientific Computing Complex at NASA Langley Research Center has been a national leader in high performance computing from the introduction of Control Data Corporation's STAR-100 in the mid-1970's. Work commencing in the mid-1980's centered around parallel vector processors such as the CRAY-2 and CRAY Y-MP. For the last decade, research at the Institute for Computer Applications in Science and Engineering (ICASE) has focused on the emerging technology of microprocessor-based parallel computing. There is general consensus that this new paradigm of computing will become a major percentage of scientific computing in the mid-1990's.

The transition to a new programming style is expected to be more difficult than the transition to vector computing in the 1970's. Parallel computers have generally immature software and require significant user effort to achieve any reasonable level of efficiency. To assist the programmer, each vendor and numerous third party software are developing parallel software tools to assist in the portability, debugging, analysis and design of parallel applications. The intent of this paper is to give users a brief introduction to the tools available on the Intel iPSC/860 (see section 1.2).

The current parallel environment at LaRC is a 32-node Intel iPSC/860 with 8 megabytes of memory per node. This machine has a hypercube interprocessor interconnect and requires an Intel i386-based System Resource Manager (SRM) as a front-end. The LaRC iPSC/860's SRM is named **Bluecrab**. There is a Sun Microsystems SPARC 1+, named **Fiddler**, that also serves as a front-end. It is not possible to login directly to the iPSC/860. By the fall of 1993, it is expected that a 66-node Intel Paragon will replace the iPSC/860. Most of the tools available on the iPSC/860 will also be available on the Paragon.

1.2 Evaluation of Parallel Software Tools

Effective parallelization of applications requires considerable effort by the programmer. This is true of the shared memory CRAY-class supercomputers as well as massively parallel computers or workstation clusters. Compilers that automatically parallelize a user's code have not matured to the level of sophistication displayed by automatic vectorizing compilers for CRAY or CONVEX class supercomputers. Debuggers and performance analysis tools for distributed memory systems are in early stages of development. However, significant work is being done by many individuals and organizations in the development of tools to assist the programmer in the design and coding of parallel code. The purpose of this paper is to summarize the effective use of some of the commercially available tools that have been installed at LaRC and to provide some evaluation as to the features that each tool possesses or lacks.

The description for each tool is organized in a similar fashion. First, an overview of the tool is provided. This section describes why the tool might be used and what functionality it provides. The next section describes accessing the tool, based on the individual vendor's documentation and the experiences of the authors. The third section is a description of how the tool is effectively utilized. This section includes examples. The following section provides observations on the ease of use of each tool and a comparison with similar tools that the authors have used on other systems. This section also discusses features that the authors feel the individual tool either lacks or needs to improve. The next section describes known problems. The final section gives references for the tool. These references may include vendor documentation or other technical papers.

The tools selected for early evaluation and testing include Intel's Interactive Parallel Debugger (IPD) and Performance Analysis Tool (PAT); ParaSoft's EXPRESS basic tool set, performance monitor and graphics tool; Applied Parallel Research's MIMDizer; and Oak Ridge National Laboratory's Parallel Virtual Machine (PVM). The paper makes extensive references to available vendor documentation and refers the reader to the *Intel Mini Manual*, written at LaRC for details on the local implementation of tools, including path names, environment variables and makefiles.

1.3 Future Direction

This work is not intended to be an exhaustive review of all parallel software. Only those tools which have been installed for use at LaRC are described in detail. The paper is intended to be dynamic. As new tools are evaluated on the Paragon or other parallel systems, their evaluations will be included in revisions or supplements to this paper. Additionally, revisions will reflect the resolution of problems, the inclusion of new features and on-going observations about the tools.

2 Express - BASIC TOOLS

2.1 Overview of Express

Express is a package of parallel software tools for FORTRAN and C applications, developed by ParaSoft Corporation for the Intel iPSC/860 and other platforms. In addition to having Express available on the Intel iPSC/860, Langley Research Center (LaRC) has a site license for both the FORTRAN and C network versions of Express on Sun Microsystems SPARCstations. Express provides a programming environment for distributed memory Multiple Instruction Multiple Data (MIMD) machines and includes a complete set of tools for developing and testing both FORTRAN and C programs. Users can develop new parallel programs by calling Express primitives from their source code. Express includes tools for message-passing, automatic data decomposition, Performance Analysis (PM), Parallel Debugger (NDB) and Parallel Graphics (PLOTIX). More information on these tools can be found in references [1] - [8] (see section 2.6). Access and other general information for Express at LaRC is located on the iPSC/860 front-end computers in the file

/ump/ipsc/share/local/express-ipsc/README.larc

Questions about Express may be directed via e-mail to:

mpp@fiddler.larc.nasa.gov

2.2 Access to Express

Express software can be run from a SUN workstation, or from an Intel 386-based microcomputer. This section includes access information from these machines at LaRC and some general information on Express software.

To access Express software, users must set the **PATH** and **MANPATH** environment variables. They may be set by executing the following statements after logging on, but before using Express on **Fiddler** or **Bluecrab.** These statements may also be added to a user's *.cshrc* file.

On **Fiddler:**

```
source /ump/ipsc/share/local/express-ipsc/setenv.sun4-fiddler
```

On **Bluecrab:**

```
source /ump/ipsc/share/local/express-ipsc/setenv.v_386-bluecrab
```

It is possible for users to access the iPSC/860 and Express software directly from their own Sun SPARCstations. To set this up, contact your workstation system administrator and the iPSC/860 system administrator at (804) 864-7474.

2.2.1 Express and NetExpress

The file in the directory

```
/ump/ipsc/share/local/express-ipsc
```

allows a user to execute Express programs on the iPSC/860.

Another version of Express, called NetExpress allows a group of networked Sun SPARCstation hosts to act as nodes, emulating the iPSC environment. NetExpress can be used to develop the initial parallel code. For more information about NetExpress, see references [5] and [6] or send e-mail to

```
mpp@fiddler.larc.nasa.gov
```

Example programs using Express are located in the

```
/ump/ipsc/share/local/express-ipsc/examples
```

directory on **Fiddler**. The examples include README files, makefiles, and *Express.cst* files that refer to specific directories and files used by ParaSoft. At LaRC the corresponding file names are:

Directory (in examples)	Directory at LaRC
/home/sampson4/express	/ump/ipsc/share/local/express-ipsc
/home/sampson4/pgi3.3/i860	/ump/ipsc/share/XDEV/i860

2.3 Detailed Description of Express

Express supports the CUBIX and host/node models of programming.

2.3.1 CUBIX Input/Output

In the CUBIX model a single program is written which is then loaded onto one or more nodes. The program starts executing on each node and communication between nodes is performed via message-passing. The two most important FORTRAN function calls in the CUBIX model are **kxinit** and **kxpara**. The function **kxinit** sets up a COMMON block and the function **kxpara** identifies the processor numbers and number of processors.

An important feature of Express is its Input/Output (I/O) system. Both loosely synchronous I/O and asynchronous I/O are supported. A loosely synchronous I/O function call is a barrier to continuance of the program. When one node makes a loosely synchronous I/O call it waits for all other nodes to make the same system call. When all nodes have made the call every node proceeds. Loosely synchronous I/O may be either single mode (**default**) or multiple mode. Asynchronous I/O function calls can be made in any node at any time regardless of the activities currently occurring in other nodes.

In single mode for input, node zero reads the input and Express broadcasts the information to all processors. For output, output from node zero is sent to the host processor. In multiple mode for input, each node receives a different piece of data. For output operations, each node outputs its own data. Both of these I/O types must be done in a loosely synchronous manner or the program will abort.

In asynchronous mode for input, each individual node can receive any piece of data. In asynchronous mode I/O statements may be executed in unpredictable order, making it hard to control. In this mode output from each node can come out in any order.

EXAMPLE (Output):

```
      Program Mtest
      Integer ndata(4)

C
C     set up Express
C
      call kxinit
C
C      identify processor numbers
C
      call kxpara (ndata )
      write ( 6,* ) 'hello world '
      call kmulti ( 6 )
      write ( 6,* ) ' I am processor ', ndata(1)
      call kflush ( 6 )
      call ksingl ( 6 )
      write ( 6,* ) ' .... and that is that! '
      stop
      end
```

If the above example is executed on 4 processors then the output is:

```
hello world
I am processor  0
I am processor  1
I am processor  2
I am processor  3
....and that is that!
```

In this example the first output is in single mode. The next four outputs are in multiple mode and the final one is in single mode. A call to **kflush** in multiple and asynchronous mode should be used to flush the buffer. Multiple mode files never flush automatically. The only way to get at the data in such a file is to call **kflush** explicitly. Common errors resulting from not flushing the buffer are the " **abort** " and " **status -1** " messages. The singular and multiple modes are not restricted to output operations, they can be applied to input operations as well.

EXAMPLE (Input):

```
      Program input
      integer ndata ( 4 )
C
C     set up Express
C
      call kxinit
C
C      identify processor numbers
C
      call kxpara  ( ndata )
C
C      get number of processors within the group
C
      nproc= ndata (1 )
C
C     default mode is single mode
C
      read (5,*) N
      write ( 6, * ) ' please enter ', nproc , ' values '
      call kmulti ( 5 )
      read ( 5, * ) I
      call kmulti  ( 6 )
      write ( 6,10 ) N,I, nproc
10    Format ( 1X,'you gave ',I3, 'and ' , I3, 'to proc ', I3 )
      stop
      end
```

When run on 2 processors with input

```
123
8,7
```

The output is

```
you gave 123 and 8 to proc 0
you gave 123 and 7 to proc 1
```

To avoid the constraint of loosely synchronous output, asynchronous mode output may be used. However, it introduces randomness to the output. Code that is guaranteed to work follows:

```
call kasync ( 6 )
if ( ... Error ... ) then
write ( 6, * ) ' we have problem '
call kflush ( 6 )
endif
```

An error condition in a particular node can be detected with asynchronous mode output. Asynchronous input is difficult to maintain since requests to read data arrive in random order.

2.3.1.1 Advantages and Disadvantages of the CUBIX Programming Model

CUBIX programs are easier to develop and maintain, since only a single program needs to be written. However, a significant drawback is the size of the code, since a copy needs to be maintained on each processor. The size of the code reduces the amount of space available for data. Another drawback is the lack of standard I/O. One of Express's three different modes needs to be used for performing I/O.

2.3.2 Host/Node Model

In the host/node model two programs must be written, one for the host and one for the nodes. The host program runs on the native host computer and the node program runs on the iPSC/860 nodes. The host computer communicates with the iPSC/860 nodes using basic Express system calls. In this model the host program may use any of the host services that were previously available to it plus the additional ones provided by Express to communicate with and control the iPSC/860. The host program allocates nodes and downloads the separately compiled node program to each processor. It also deallocates nodes after the node program finishes execution. All I/O must be handled by the host program. The interface or control portion of the code remains on the host. The interface between host and node program is provided by Express function calls which allow data to be transferred between host and nodes. The compute-intensive portion of the application is executed on the iPSC/860 nodes. The Express FORTRAN function calls **kxinit**, **kxopen**, and **kxload** are used in the host program to setup Express, to allocate processor groups, and to load the node program into each processor.

The use of byte-swapping calls is needed when sending messages between the host and the iPSC/860 when the remote workstation does not use Intel's byte ordering convention. The Intel convention specifies that the least significant byte of an integer is stored at the lowest memory address. Sun SPARCstations, such as **Fiddler**, use a different byte ordering convention. Byte-swapping calls are provided by Express.

2.3.2.1 Advantages and Disadvantages of the Host/Node Model

In the host/node model a portion of the code is kept on the host computer. This makes more memory available on the iPSC/860 nodes. The host can keep machine specific code. For example, if considerable amount of time

has been spent developing a complex graphical user interface, it may be wasteful to attempt porting it to the iPSC/860 environment.

Debugging host/node model programs, however, is difficult, since nodes are unable to perform their own I/O without coordination with the host. So to debug with print statements one has to change both the host and node programs. It is also more time consuming to develop and maintain host/node model programs.

2.3.3 Compiling CUBIX and Host/Node Programs

Express supports FORTRAN and C. The FORTRAN compiler is *xf77* and the C compiler is *xcc*. For more information on the CUBIX and the host/node programming model consult references [1] and [2].

The FORTRAN compiler for Express is:

xf77 [-c] [-o outname] [-kXN|-kXH|-kcubix] [-v] files....

Description of options:

-c	Compile only - do not proceed to link resulting object files.
-kcubix	Use CUBIX programming model.
-Mperf	Enables profiling.
-kXH	Use host/node programming model for a host program.
-kXN	Use host/node programming model for a node program.
-o outname	Specify an alternate name for the executable program produced by the linker. **Default is 'a.out'.**
-g	For correct functioning of profiling tool *xtool*.

The C compiler for Express is:

xcc [c] [-o outname] [-Dname[=value]] [-Idirname] [-kXN|-kXN|-kcubix] [-v] files....

Description of options:

-c	Compile only - do not proceed to link resulting object files.
-Dname **-Dname=value**	Define preprocessor symbol and optionally assign a value. It has the same effect as using '#define name value' in program line.
-Idirectory	Add a directory to the path for include files.

-kcubix	Use CUBIX programming model.
-Mperf	Enables profiling.
-kXH	Use host/node programming model for a host program.
-kXN	Use host/node programming model for a node program.
-o outname	Specify an alternate name for the executable program produced by the linker. **Default is 'a.out'.**
-v	Generate listing of all command lines before they are executed.
-g	For correct functioning of profiling tools.

2.3.4 Executing Express CUBIX Programs

The basic syntax for the *cubix* command is

cubix [-n nodes] [-d doc] [-t time] [-D] [-mc|x|e] prog [arg1][arg2]]..

This command provides an interface between the host/node applications and the host file system and operating system utilities. It is also responsible for node application and the communication of command line arguments to a node program.

Description of options:

-n nodes	Allocate nodes for this process. **Default is 1.**
-d doc	Alternative to -n switch. Specify size of processor group logarithmically in manner suitable for hypercubes (i.e.,doc=0 for 1 node,doc=1 for two,doc=2 for four etc). **Default is 0.**
-t time	Time out the process after the given number of seconds. This can be useful in detecting hung programs. **The default is no time out.**
-m[c|x|e]	Enable the performance monitoring tools. The 'c','x',and 'e' characters refer to the communication, execution and event driven profiling systems respectively and may be combined.

2.3.4.1 Sample CUBIX and Host/Node Programs

EXAMPLE (Interactive CUBIX Program):

```
      Program hello
      integer nddata(4),procnum, nprocs
C
C Express cliche
C
      common/XPRESS/NOCARE,NORDER,NONODE,IHOST,IALNODE,IALPRC
C
C Start up Express, find out processor numbers and number of processors
C
      call kxinit
      call kxpara(nddata)
      procnum=nddata(1)
      nprocs=nddata(2)
C
C Write same thing from each processor; it is written once.
C Then read a single value  C  into each node.
C
      write(6,1)  nprocs
1     format(1x,'Hello, there are ',i3,'nodes,give me a number: '
      read (5,*) ival
C
C  Write from each processor;this appears in processor number order.
C
      call kmulti(6)
      write(6,2) procnum, procnum,ival,ival*procnum
2     format(1x,'I am processor ',i6, ':', i6,' x ' , i6,' = ', i6*i6)
      call kflush(6)
C
C  Switch back to single mode, prompt for one value for each node,
C   read them in processor number order
C
      call ksingl(6)
      write(6,3) nprocs
3     format (1x,'Please enter ',i4, ' values, one for each  node')
      do 10 i=1,nprocs
      read(5,*) jval
      if ( i.eq.procnum+1) ival = jval
10    continue
C
C  Write the values asynchronously;  in random order. Note that it is often hard
C  to tell on small number of nodes the difference between multi and async modes
C
      write(6,*) 'The numbers you entered, in random order:'
      call kasync(6)
      write(6,4) procnum,ival
4     format (1x,'Node ',i4, ' received ', i4)
      call kflush(6)
      stop
      end
```

A sample makefile for this CUBIX program follows:

```
###start of makefile for Intel Express CUBIX FORTRAN code
all:ipsc
ipsc:hello
#
hello:
	xf77
runipsc:
	cubix
clean:
# ipsc code
hello : hello.f
xf77 -o $@ hello.f -kcubix
# load executable 'hello' on 4 processors.
runipsc:   cubix -n4 hello
clean: -rm -f *.o *_fpp.f
```

To compile and execute using this makefile enter the following (Note that all **boldface** type is the response from the computer):

```
% make all
xf77 -o hello hello.f -kcubix
% make runipsc
cubix -n4 hello
cubix version 3.2.4 -- copyright (c) 1991 ParaSoft
Allocated 4 nodes, origin at 0, process id 3048.
Loading "hello" into all processors....
Loaded,starting...
Hello, there are 4 nodes, give me a number:
<  2
I am processor 0 : 0X   2 = 0
I am processor 1 : 1X   2 = 2
I am processor 2 : 2X   2 = 4
I am processor 3 : 3X   2 = 6
please enter 4 values, one for each node :
<  1
<  2
<  3
<  4
The numbers you entered, in random order:
Node 0 received 1
Node 1 received 2
Node 2 received 3
Node 3 received 4
Execution Terminated:
System 0:24    user 3:11
CUBIX : exit status 0
%
```

EXAMPLE (Interactive Host/Node Programs):

Host program:

```
      Program host
      common /XPRESS/ NOCARE,NORDER,NONODE,IHOST,IALNOD,IALPRC
      integer fromnd(2),ntimes
      integer check,k,status
      integer pgind,nodes
      integer src,type,swap
      integer env(4)
      character*80 prgnam,device
      parameter( device=' /dev/transputer',prgnam='node')
      data type /123/
C
C Start up Express. This must be the first system call you used.
C Read the number of nodes. Also find out whether or not to stop the program
C upon loading. This is useful for debugging. This is done by having a negative
C number of nodes mean "stop". Kxpaus call stops a program at its first
C instruction after being loaded. Kxtswa is a byte-swapping routine
C
C
C START
C
      CALL KXINIT
      swap=kxtswa()
      write(6,*) 'number of nodes?'
      read(5,*) nodes
      if(nodes.lt.0)then
      nodes=-nodes
      call kxpaus
      endif
C
C Allocate a processor group containing nodes processor in the device pointed
C to by the character device.
C
      pgind = kxopen(device,nodes,nocare)
      if(pgind.lt.0) stop 'failed to allocate nodes'
C
C load the program
C
      status = kxload (pgind,prgnam)
      if(status.lt.0)then
      write(6,*)'failed to load program'
      call kxclos(pgind)
      stop
      endif
C
C get system parameters and construct a checksum to compare with the
C values returned from the cube
C
      status=kxpara(env)
      check=0
      do 10 k= 1,env(2)
        check=check + (k-1)
10    continue
C
```

```
C Prompt for the number of times to pass the message around a ring
C
      write(6,*) 'how many times should the message go around'
C
      read(5,*) ntimes
C
C Send the count to the cube and then read back a message for each cycle
C This is rather tricky on machines with reversed byte orders. Therefore
C byte-swapping is needed to send them to the nodes,and then swap them
C
      if (swap.eq.1) call kxswaw(ntimes,ntimes,4)
      status=kxbrod(ntimes,ihost,4,ialnod,dummy,type)
      do 20 k=1,ntimes
        src=0
        status=kxread(fromnd,8,src,type)
        if(swap.eq.1) call kxswaw(fromnd,fromnd,8)
        if (fromnd(1).ne.env(2).or.fromnd(2).ne.check) then
          write(6,*) 'error in node communication'
          write(6,*) 'expecting ',env(2),check
          write(6,*) 'received',fromnd(1),fromnd(2)
          call kxclos(pgind)
          stop
        else
          if(mod(k,100).eq.0)then
            write(6,*)'done',k
          endif
        endif
20    continue
      write (6,*) 'finished'
      call kxclos(pgind)
      stop
      end
```

Node program:

```
      Program Node
      common /XPRESS/ NOCARE,NORDER,NONODE,IHOST,IALNOD,IALPRC
      integer indat(256),outdat(256)
      integer tohost(2),ntimes
      integer env(5)
      integer k,nshift,status,dummy
      integer fwdnod,bcknod
      integer type,dest,nprocs
      integer fadd
      common /junk/ idata(100)
      external fadd
      integer swap
      data type/123/
C
C  start up Express. This must be the first Express system call used.
C
      call kxinit
      swap=kxtswa()
C
C read system parameters,number of nodes etc....
C
      status=kxpara(env)
C
C  Now set up the channels to use in the ring. Map a one dimensional chain
C of processors onto the 'doc'dimensional hypercube grid with (1<<doc)
C processors in it
C
      nprocs=env(2)
      status=kxgdin(1,nprocs)
      fwdnod=kxgdno(env(1),0,1)
      backnod=kxgdno(env(1),0,-1)
C
C now read the number of iterations from the host - note that the number
C of forwarding operations is this parameters times the length of the ring
C
      idata(1)=123
      idata(2)=type
      idata(3)=-1
      status=kxbrod(ntimes,ihost,4,ialnod,dummy,type)
      if (swap.eq.1) call kxswaw(ntimes,ntimes,4)
      idata(3)=ntimes
      do 10 k=1,ntimes
C
C shift data around the ring we just set up
C
        do 20 nshift=1,env(2)
          if (swap.eq.1) call kxswaw(outdat,outdat,512)
          status=kxchan(indat,512,bcknod,type,outdat,512,fwdnod,type)
          if (swap.eq.1) call kxswaw(indat,indat,512)
20      continue
C
C Now send a message to the host. Add up a bunch of ones and also
C our processor numbers
C
        tohost(1)=1
        tohost(2)=env(1)
```

```
            if(swap.eq.1) call kxswaw(tohost,tohost,4*2)
            status=kxcomb(tohost,fadd,4,2,ialnod,dummy,type)
            if (swap.eq.1) call kxswaw(tohost,tohost,4*2)
            if(env(1).eq.0)then
              if(swap.eq.1) call kxswaw(tohost,tohost,8)
              status=kxwrit(tohost,8,ihost,type)
            endif
10        continue
          call kxexit
          stop
          end
C
C User supplied summing function to be used in kxcomb call
C
          integer function fadd(i,j,size)
          integer i,j,size,fadd
          i=i+j
          fadd=1
          if(swap.eq.1) call kxswaw(i,i,4)
          if(swap.eq1) call kxswaw(j,j,4)
          return
          end
```

A sample makefile for this host/node model program follows:

```
all:ipsc
ipsc:host node
# ipsc code
host:host.f
        xf77 -kXH -o $@ host.f
node:node.f
        xf77 -kXN -o $@ node.f
# run host-node code on 4 processors
runipsc:ipsc
         host 4 node
clean: -rm -f host node *.o core
```

To compile and execute host/node code enter the following (Note that **boldface** type is output from the computer):

```
% make all
xf77 -kXH -o host host.f
host.f:
MAIN exsamp:
xf77 -kXN -o node node.f
% make runipsc
host 4 nodes
number of nodes?
< 4
Allocated 4 nodes, origin st 0,process id 3165
Loading "node" into  all processors...
Loaded, starting ...
How many time should the message go around?
< 2
Finished
%
```

2.3.5 Express System Calls

The Express system calls are used from within CUBIX and host/node programs. Express offers a variety of system calls. In this section the basic calls for programming with Express will be described.

The Express system calls are divided functionally into seven groups: system initialization, processor allocation and cube control, message-passing, global operation, data decomposition, I/O, and utility functions. There is a domain associated with each system call. The system calls listed in the following section also give the domain in which they can be invoked. Some system calls can be issued by either host or node programs others are available only to host or only to node programs.

2.3.5.1 System Initialization

The **kxinit** routine must be the first Express routine called in both host and node programs. It serves to initialize the internal state of Express and also to set up a common block containing useful parameters for use by application code.

Systam call	Domain	Description
kxinit	host/node	starts Express system.

2.3.5.2 Processor Allocation and Cube Control

The following calls are to be issued from the host program only. The **kxploa** call provides a complementary interface to the **kxload** routine for loading programs into groups of processors. Instead of loading the entire array with a single node program, this routine allows different applications to be loaded into individual nodes of the machine. The **kxopen** call must be used before attempting to access any processor groups. The **kxclos** call is used at the end of the application for releasing a cube.

System call	Domain	Description
kxclos	host	deallocate processors.
kxload	host	load a program into nodes.
kxopen	host	allocate a processor group.
kxpaus	host	arrange for programs to be loaded stopped.
kxploa	host	load a program into individual nodes.

2.3.5.3 Data Decomposition

The **kxpara** routine is used to determine runtime configuration. The information of runtime environment is returned to the array elements of **kxpara**. The first array element returns node id of calling process and the second array element returns number of processors allocated in the processor groups.

Other data decomposition tools collectively fall under KXGRID routines (KXGRID is not a Fortran-callable routine), whose routines include functions to perform automatic decompositions of user domains onto the underlying machine topology. A user specification for a problem domain which has the topology of a Cartesian grid in N dimensions is mapped onto the hardware topology. A partial list of the KXGRID routines is listed. More information can be found in references [1] and [2] (section 2.6).

System call	Domain	Description
kxpara	host/node	determines runtime configuration.
kxgdin	node	performs elementary mapping and must be called before any of the other kxgrid routines except kxgdsp.

System call	Domain	Description
kxgdsp	node	used to divide up processors between user specified dimension.
kxgdsi	node	used to distribute an array over user grid. It distributes global index over processors and assigns global index to local index.

2.3.5.4 Message-Passing

The 32-node Intel iPSC/860 at Langley is a scalable distributed memory parallel supercomputer. Communication between processors occurs through message-passing. Messages can be synchronous or asynchronous and are characterized by a message length in bytes, a type and a status. The accepted message is read into a receive buffer. The type is an identifier, determined by the programmer allowing control and validation of messages by type. The status argument is a mechanism by which one can check for the completion of asynchronous messages. Express provides system calls for: synchronous message-passing, synchronous vector message-passing, test for an incoming message-non-blocking, asynchronous message-passing.

A synchronous send indicates that the submitting process waits until the send is complete. The completion of the send is not a verification that the message is received. The send function **kxwrit** returns the number of bytes written, or -1 in case of unrecoverable errors. A synchronous receive blocks the receiving process until a message with suitable parameters has arrived. The vector send and receive functions allow send and receive of non-contiguous blocks of data. This is the main difference between vector and basic send and receive functions. A list of frequently used message-passing functions follows.

System call	Domain	Description
kxread	host/node	receive a message and wait for completion.
kxwrit	host/node	send a message and wait for completion.
kxvrea	host/node	receive non-contiguous blocks of data.
kxvwri	host/node	send non-contiguous blocks of data.

2.3.5.5 Global Operations

Global operations provide a high-level construct for communication among node processors. A list of frequently used global operation functions follows.

System call	Domain	Description
kxbrod	host/node	performs broadcasting operation among the processors. This function broadcasts n-bytes of data from an from a origin in the indicated buffer. The broadcast is to all processors when the argument nnodes is set to IALNOD. IALNOD is defined in the XPRESS common block and is set up by the kxinit function.
kxchan	host/node	performs a similar task to a pair of successive calls to kxwrit and kxread. User is freed from worrying about the order of sends and receives.
kxcomb	node	The kxcomb node is used to perform "combining" operations on data within the node processors. The operation to be combined is supplied by the user. In case of an error, -1 is returned.
kxsync	node	implements synchronization points in an application. It is guaranteed that no processor will proceed past the call to kxsync until all are ready.

2.3.5.6 I/O

A partial list of I/O functions is described below.

System call	Domain	Description
kmulti	node	Each node can read its own data. Output requests also can be made independently. Call to kflush must be used to flush buffers else errors will occur. Most common errors are "abort" and"status -1". Calls to kmulti must be made in loosely synchronous fashion.

System call	Domain	Description
ksingl	node	for read operation node 0 reads and Express broadcasts the same data to all other nodes. For write operation only node 0 transmits. Calls to ksingl must be made in loosely synchronous fashion. **This is the default mode.**
kmwrit	node	writes independent data from each node to the file indicated by the argument.
kasync	node	I/O requests are handled independently on the processors. No interprocessor synchronization is performed.

2.3.5.7 Utility Routines

Utility routines include byte-swapping routines and timing routines. The use of byte-swapping calls are needed when sending messages between the cube and the host on which the remote workstation does not use Intel's byte-swapping convention. Intel stores the least significant byte of a word at the lowest memory address. Sun SPARCstations, such as **Fiddler**, use a different byte ordering convention. They store the most significant byte at the lowest address. The consequence of this is that byte-swapping must be performed when communicating data between the host processor on **Fiddler** and node processors on the iPSC/860. Byte-swapping calls require three arguments. The first is the buffer from which data is taken and second is the buffer into which it should be placed after swapping. The last argument is the number of bytes in the buffer to be swapped. Following is a partial list of utility routines.

System call	Domain	Description
kxtick	node	returns time in microseconds.
kxswaw	host/node	reverses the bytes in 4 bytes quantities.
kxtswa	host/node	checks whether bytes need to be reversed. If the returned value is 1, then bytes need to be reversed.

2.4 Observations on Using Express

In order to calculate communication overhead of basic message-passing calls relative to an iPSC version, an iPSC version "ring" program and a Express version "ring" program may be executed. The Express version of the "ring" program performs timing analysis to compute overhead associated with **kxread** and **kxwrit** functions compared to Intel's **csend** and **crecv** functions. The "ring" program circulates a message among all processors which are conceptually arranged in a ring configuration. Each processor receives a message from its predecessor and forwards it to its successor. The cycle is repeated a hundred times for each message length. Message length varies from 0 to 32768 bytes.

On average, Express overhead was about 20% more costly than the native Intel message-passing calls. On the other hand, for one of the NAS parallel benchmarks, Express' combining function **kxcomb** was less expensive than Intel's **gdsum** function. Also for another benchmark code Express's timing using **kxrecv** and **kxsend** showed improved performance over Intel's **irecv** and **csend.**

The difference between **kxread/kxwrit** and **kxchan** is basically one of safety. If messages are sent with **kxwrit** and then read with **kxread,** it is implicitly assumed that the low lying message-passing system has enough memory to buffer the messages until it is read. The **kxchan** breaks up a large message into small pieces and arranges its proper delivery by alternating reads and writes internally.

For short messages, therefore, **kxread/kxwrit** will usually result in improved performance but might scale badly. As more nodes are added, there is a tendency to dump more messages into the implied buffer pool until there is no more memory. The performance of **kxchan** can be improved markedly by tuning the Express packet size to match the communication size. This is done by modifying the value of the NBSIZE variable in the Express customization file. If messages sent are larger than NBSIZE then Express tries to dynamically allocate a larger buffer which adds overhead.

The routine **kxcomb** can be used in two ways. The arguments "size" and "items" specify size of data items in bytes and the number of data items to be combined respectively. For performance reasons it is important to know how these arguments relate to each other and performance. If a user wants to sum up 6 integer numbers in each processor, there are two ways to invoke the combine routine in Express. The first likely invocation is to specify: size=4 and number=6. In this case the function will be called separately for each item and the program will run slowly. The other possible invocation is: size=24 and number=1. In this case the routine will be invoked only once and will have to perform array summation. Programs run faster in the latter case. Also since Express's combine operation allows users to write their own combining function, it has greater functionality than Intel's set of routines.

Even though the manual mentions that the timing function **kxtick** returns the number of hardware clock ticks, in reality it returns time in microseconds. For many users Express's loosely synchronous mode of I/O creates inconvenience. Currently ParaSoft is trying to modify I/O operations for their next version. In the next version the default mode will be native I/O and users will have to switch mode to get Express's I/O. Express manuals do not provide diagnostic information for system call errors.

The main strength of Express is its portability. Express can be used on a variety of architectures including a network of Suns, CRAY Y-MP, iPSC/860 etc. This offers portability across many systems. According to ParaSoft Express is 100% portable across different architectures. This should make it possible for users to perform initial code development, testing, and debugging on their own workstations, thereby reducing much of the development activity from the iPSC/860.

Given that Express runs on a variety of platforms, the possibility of running a single Express program on a combination of them is feasible. A software component called "glueworks" can provide the multi-platform capability.

2.5 Known Bugs of Express

The timing function **kxtick** does not return the number of hardware clock ticks as described in the manual. The routine **kxtick** returns time in microseconds.

2.6 References for Express

[1] Express FORTRAN Reference Guide
[2] Express C Reference Guide
[3] Express FORTRAN User's Guide
[4] Express C User's Guide
[5] Express FORTRAN Language Introductory Guide for Workstations
[6] Express C Language Introductory Guide for Workstations
[7] Express FORTRAN Language Introductory Guide for iPSC/860
[8] Express C Language Introductory Guide for iPSC/860

3 Express - Performance Analysis Tool (PM)

3.1 Overview of PM

The PM utilities are post-mortem tools for analyzing program execution, communication performance and event traced performance of application programs on the iPSC/860 system. Data is accumulated during the execution of the user program and then analyzed off-line, after execution has completed. This data can be analyzed graphically or in the tabular forms with Sunview or the X Window System (X).

The PM profiling data can be gathered in two ways. Profiling data may be collected automatically using switches or through a programmatic interface. The use of compiler switches to automatically gather data with the FORTRAN and C compilers is the easiest and preferred method. The programmatic interface method of gathering profiling and tracing data requires specific changes to the source code before the code is compiled. Since this method requires additional work, it is recommended that it be used when a specific problem needs to be isolated.

3.2 Access to PM

Access and general information on Express software is described in Chapter 2, section 2.2.

3.3 Detailed Description of PM

The PM tools utilities are designed to analyze Express parallel performance. PM consists of three tools: *ctool*, *xtool* and *etool*. These tools are used to analyze performance of application programs.

3.3.1 Using the Program Execution Tool (xtool)

The execution profiler tool, *xtool*, monitors time spent in individual routines. This allows the user to focus attention on the most time consuming areas which would benefit most by improvements.

The execution profiler relies on data contained in the symbol table for proper functioning. This is done by specifying the *'-g'* switch during the linking phase of the program.

The system works on a statistical principle. Every few milliseconds a system routine runs which looks at the current instruction being executed in the user application and increments a counter noting the memory addresses. In this way it builds a histogram of the frequencies of hits in various areas of the program, and determines the amount of time spent in particular routines.

3.3.2 Using the Program Communication (ctool)

The communication profiler tool, *ctool*, assesses time spent in communication and I/O. On each node, data is accumulated to measure the following:

* Time spent calculating, communicating between processors and performing I/O functions. This leads to an estimate of program overhead and efficiency.

* Total numbers of calls to the communication system. It provides a simple estimate of load imbalances.

The following function information is recorded, on each node:

* Number of calls to each individual function.

* Distribution of return values from each function (i.e. message length read, message length written, number of objects broadcast etc.)

3.3.3 Using the Program Events (etool)

The event profiler tool, *etool*, shows the interactions between nodes and allows user specified events to be monitored. The goal is to provide a detailed examination of the interaction between various nodes as time progresses. The following event information can be recorded:

* The time at which the event occurred.

* An index value indicating the nature of the event.

* A program variable with its value at the time of the event.

3.3.4 Compiling and Executing a CUBIX PM Program

Profiling information on the Fortran CUBIX program **hello** in section 2.3.5 can be gathered automatically by using the *-Mperf* switch during compilation.

```
xf77 -Mperf -o  -g hello hello.f -kcubix
```

To turn on communication, execution, and event profiling systems the *-mcxe* switch is used during execution.

```
cubix -n4 -mcxe hello
```

The CUBIX command with the *-mcxe* switch creates log files named *cbxprof.cp, cbxprof.xp, cbxprof.ep* to be analyzed by *ctool, xtool* and *etool* respectively. The syntax to execute each tool is:

```
xtool[options] program_name log_file_name
etool[options] program_name log_file_name
ctool[options] program_name log_file_name
```

where *program_name* specified the name of the program from which the profile data is gathered. The *log_file_name* specifies the name of the file containing the profile data that the PM tools examine.

A partial list of options follows:

-p suppress graphical output.Tabular output is sent to stdout.

-T Use an alternative graphical device for output.

The *man* page for each tool has a more complete description.

Here are examples for using the *-p* option for tabular output and *-T* option for viewing data in graphical form using the X environment.

```
ctool -p program_name log_file_name
ctool -TX program_name log_file_name
```

References [1] - [4] (section 3.6) have more information on PM.

3.4 Observations on Using PM

3.4.1 Data Gathering Using Compiler Switches

Advantages: Using compiler driven switches is the easiest, most direct way of gathering performance data for the PM utilities. The compiler *-Mperf* option, enables data gathering for any or all PM utilities. This is a preferred way to gather performance data.

Disadvantages: While the automatic method is easy to implement, it does have the following disadvantages.

* It is relatively broadbanded, i.e., the entire program is profiled instead of a possibly smaller code segment that may be of interest.

* With the automatic method, the amount or type of information gathered can not be controlled. For example, a user may only be interested in profiling data for iterations 110-125 of a 1000 iteration loop. This cannot be achieved with the automatic method.

* The automatic method of gathering data may have an undesired impact on program performance. Because each procedure/subroutine has at least one entry and one exit system call added to it, programs with many procedures/subroutines can experience a significant growth in processing time and thus a degradation of program performance.

3.4.2 Data Gathering Using the Programmatic Interface

Advantages: This method allows the user more control over the amount and type of data that is collected.

Disadvantages: Programmers using the manual method of collecting profiling data must have a good knowledge of the code that PM is profiling. One way of gaining this knowledge is to first use the automatic method to get a feel for the general performance of the overall program, and then use the manual method to examine specific areas in more detail.

3.5 Known Bugs of PM

No bugs have been found at this point.

3.6 References for PM

[1] Express FORTRAN Reference Guide
[2] Express C Reference Guide
[3] Express FORTRAN User's Guide
[4] Express C User's Guide

4 Express - Graphics Tool (PLOTIX)

4.1 Overview of PLOTIX

PLOTIX is an extension of the Express CUBIX model which functions as a parallel graphics system for supported parallel and distributed systems. PLOTIX provides some basic graphics primitives (points, lines, polygons, etc.) with associated attributes (color, line style, fill pattern, etc.); supports normal windowing, viewporting, and clipping; provides for rudimentary (locator) graphics input; and supplies a two dimensional contouring facility for rectangular and irregular regions. Device-independent graphics output is provided with supported devices including the X Window System (X) and PostScript (among others). The PLOTIX graphics system is described in references [1] - [2] (section 4.6).

4.2 Access to PLOTIX

Since PLOTIX is an extension of the Express CUBIX model, consult section 2.2 for detailed access information. Analogous to the CUBIX model, there are sample FORTRAN and C PLOTIX programs in the subdirectory:

```
/ump/ipsc/share/local/express-ipsc/examples/plotix
```

A PLOTIX program is compiled and linked (in FORTRAN or C) exactly as a CUBIX program with the additional inclusion of the PLOTIX library. Therefore, using the earlier CUBIX notation, a PLOTIX FORTRAN program is compiled and linked using

```
xf77 [-o program] ... -kplotix files ...
```

and a C program may be compiled and linked using

```
xcc [-o program] ... -kplotix files ...
```

A PLOTIX program is executed as a CUBIX program with the addition of the selection of the appropriate device driver using the *"-T"* command line option. Again using the earlier CUBIX notation, a PLOTIX program execution command line using the X driver is

```
cubix [-n nodes] ... -TX program ...
```

and using the PostScript driver is

```
cubix [-n nodes] ... -Tps program ...
```

Other supported devices are linked in a similar manner.

4.3 Detailed Description of PLOTIX

PLOTIX uses a variation of the CUBIX programming model. Consequently, all Express FORTRAN (C) functions, including the CUBIX-specific functions, are supported under PLOTIX. As with other Express functions, the user should check the return codes from all PLOTIX functions calls before proceeding. The following sections give the FORTRAN function name followed by the C function name in parenthesis.

The PLOTIX graphics system must be initialized with the FORTRAN (C) function **kopenp(openpl)** and closed with **kclosp(closepl)**. PLOTIX buffers all graphics output and the user is responsible for both declaring and flushing the graphics buffer. The user may wish to experiment with the size of the buffer which is supplied as an argument to **kopenp(openpl)** (a default size is provided). The user must flush the graphics buffer before any graphics output is displayed. Analogous to the CUBIX I/O modes (see section 2.3), the graphics buffer may be flushed in single-mode, multi-mode or asynchronous-mode using **ksendp(sendplot)**, **kusend(usendplot)** or **kasendp(asendplot)**, respectively, conforming to the loosely synchronous versus asynchronous conventions.

PLOTIX utilizes normalized coordinates in the range [0.,1.] to represent screen-space or view-space. The user may partition this space with any number of viewports with only one viewport being active at a time. The user may define world-space or window-space based on the range of the application data. The parallel graphics system affords the capability of clipping the physical data at the window boundaries prior to display using function **ksetcl(setclip)**.

Device-independence is an important attribute of PLOTIX. In support of this feature, PLOTIX provides a function **kpxgop(getxplot)** which permits run-time determination of device-specific characteristics (such as support for color or graphics input) for the purpose of generating code that is conditionally executed based on these retrieved characteristics.

The supplied contouring system provides some high-level functionality within PLOTIX. Function **kcntor(contour)** generates a two-dimensional

contour map over a rectangular region based on data computed from a user-supplied function (as opposed to a user-supplied two dimensional array of values). In the case of irregular regions, a combination of the lower-level functions **kintl(initlevel)** and **kgetpt(getpoint)** are used in conjunction with the supported polygon drawing routines and the user supplied function for calculating contour values. Examples on the use of these contour routines and other PLOTIX functions are provided in the aforementioned documents and on-line as addressed earlier in this section. See section 4.6 for references.

4.4 Observations on Using PLOTIX

PLOTIX provides an architecture and device independent system for performing parallel graphics. At the present time, the functionality of PLOTIX is fairly limited, but the supplied basic primitives could be combined by the user to construct more complex graphics applications. The PLOTIX contouring system provides the only high-level graphics capability supplied with the system. Built-in visualization capabilities for hidden line/surface rendering, three dimensional contouring, isosurfaces, vector display, image operations, are not available.

There is little with which to compare the performance of the Express PLOTIX model, except to note that since PLOTIX is an extension of the CUBIX I/O model, PLOTIX programs inherit the advantages and disadvantages of a CUBIX program. From a user perspective, the interface to PLOTIX is relatively straightforward implying that parallel graphics applications may be constructed with a minimum number of function calls.

4.5 Known Bugs of PLOTIX

Although references [3] and [4] provide an alphabetized description of PLOTIX routines. It is sometimes difficult to find individual routine descriptions because many descriptors are for groups of routines.

4.6 References for PLOTIX

[1] Express FORTRAN User's Guide
[2] Express C User's Guide
[3] Express FORTRAN Reference Guide
[4] Express C Reference Guide

5 Interactive Parallel Debugger (IPD)

5.1 Overview of IPD

IPD is a symbolic source-level debugger for Fortran, C and assembly language programs running on the Intel iPSC/2 and iPSC/860 Concurrent Supercomputers. IPD contains most of the features of current debuggers designed for serial programs and adds custom features supporting the parallel programming model. IPD contains debug environment commands which control the debugging environment. Another class of IPD commands gives the user precise control over program execution. A third class of commands perform program examination and modification. IPD is perfectly capable of debugging in a true MIMD (Multiple Instruction Multiple Data) environment where different programs may be on the nodes but for the purpose of this discussion only the prevalent SPMD (Single Program Multiple Data) case is addressed. More information on this debugging tool can be found in references [1] and [2] (see section 5.6). Information regarding access for IPD is located in the file

```
/ump/ipsc/share/local/IPD-ipsc/README
```

Questions about IPD may be directed via e-mail to :

```
mpp@fiddler.larc.nasa.gov
```

5.2 Access to IPD

IPD assists in debugging only node programs resident on the iPSC/860. IPD is invoked and commands entered only from the SRM (**Bluecrab**). If you're logged onto a system remote host you must log into **Bluecrab** in order to be able to utilize IPD. All IPD commands must be entered from **Bluecrab.**

5.2.1 Invoking IPD from Bluecrab

Before using IPD, the node program must be created using the *"-g"* compile option, and the cube allocated. The IPD invocation syntax is

```
ipd
```

If the file *.ipdrc* exists in your home directory, IPD will automatically execute commands in this configuration file at this time. The node

program to be debugged is loaded onto the appropriate nodes via the IPD *load* command and execution commences.

5.2.2 Node Program Specification

An application program is loaded onto the nodes and "into" the debugger only by way of the *load* command (see 5.3.3).

5.2.3 Exclusive IPD Features

IPD uses the concept of "debug context" to implicitly (by default) or explicitly denote which nodes are the target(s) of IPD commands. This is the one mechanism which promotes debugging asynchronously executing node programs on the cube.

Another feature is the display of the send/receive messages queues that contain outstanding messages of types sent but unreceived and pending receives.

Since node program execution synchronization is not guaranteed even for the SPMD model, the process command displays the present state of each node program as a significant diagnostic feature.

5.3 Detailed Description of IPD

The following represents the typical scenario of a debugging session under IPD with particular emphasis on the custom parallel debugging features which IPD affords. A single FORTRAN program is assumed (SPMD model).

5.3.1 Node Program Compilation

Normally, compilations for the Intel iPSC/860 use the *if77* cross-compiler on **Fiddler.** A sample *if77* command line syntax looks like

```
if77 -o node -g myprog.f -node
```

where *-g* gains access to the symbolic debug information at level 0 (**default**).

5.3.2 Other Preparation

The user can store commands, normally entered immediately after IPD invocation, into the *.ipdrc* configuration file. Examples here include default aliases, breakpoints from prior sessions, other aliases to display known variables at breakpoints, default context, loading of node program(s), execution commencement, etc.

5.3.3 Asynchronous Node Program Execution Monitoring

An assumption here is that the **load** command is executed to store the SPMD model onto the cube (nodes) and into the debugger. Also, the initial **run** command starts the execution of the FORTRAN program resident on all nodes.

From this point forward, IPD is called upon to be able to efficiently and effectively deal with every possibility that might occur. The **wait** command is one mechanism IPD has to deal with the fairly normal occurrence of all nodes running to completion or breakpoint. This command can ensure synchronization of nodes when debugging from breakpoint to breakpoint. Breakpoints are usually set at the same line number for all node programs. This technique however is far from foolproof due to the possibilities of:

1. program logic errors (infinite loops)
2. extremely slow executing code due to other logic problems (semi-infinite looping, unexpected data values, etc.)
3. aberrant data values causing abnormal conditions in one or more separate nodes.

The user can interrupt (Ctrl-C) the executing node programs at any time. Following this, the user should issue the **stop** command, followed by the **process** command, to determine the exact status of all node programs through the display of any informatives.

5.3.4 Some Diagnostic Approaches at Halt

The **process** command (see 5.3.3) is a high-level command reporting overall information regarding the state of each node. This may expose an anomaly in program logic for all nodes or point to specific nodes as differing from the majority.

Also to be considered as a high-level command in the command hierarchy is the message passing queue commands. There shouldn't be any outstanding sends/receives at synchronization points! Problem resolution here depends on the user's ability to know what should be happening in the message passing system of the program and when it should occur. Immediately below the high-level commands comes the **frame** command, which is used to fix the order of routine invocation so that the user might also know the exact halting location as well as the specific call chain in force.

Below the frame level, IPD has two fairly simple commands to examine (**display**) and modify (**assign**) program variables. Incorrect or unexpected values in program variables usually signals the experienced debugger to discover the error source. Many times, entering the same sequence of commands is facilitated by the **exec** command, which automatically or semi-automatically causes each IPD command stored on a file to be executed.

5.4 Observations on IPD

In general, IPD seems fairly easy to use and doesn't contain any difficult concepts to grasp which would inhibit a relative novice. Probably the most difficult situation encountered is interpreting correctly the output from the **frame, process** and outstanding message queue commands. The novice user is likewise not burdened down with making sense of a great many commands to comprehend. This, however, can be disconcerting to the experienced user expecting to find more powerful commands presently available in current (UNIX) interactive debuggers. IPD may be regarded as a bare bones debugger, but one that should suffice given the task of debugging parallel applications programs on the Intel iPSC/860. The reasoning behind these general observations regarding IPD will be delineated in terms of it's relative strengths and weaknesses from the perspective of other debugging systems, both serial and parallel.

5.4.1 Strengths

IPD's strong suit is the use of debug context. The user is well informed at all times about which nodes are affected by any IPD command. The context command allows the user to change the default debug context should conditions warrant. The outstanding message queue commands are invaluable in helping diagnose problems due to mismatched/untimely communication patterns. Together with the **process** command, they provide a high-level view of the program's state.

Breakpoint setting is also effectively implemented, especially in reference to the data breakpoint option. This can be quite handy, but notoriously slow if the user hasn't sufficiently narrowed down the code of interest. The **step** command is effective when suspected code is narrowed and when one or several nodes are suspected of not taking the same logic path as the majority or when bad/unsuspected data is encountered.

5.4.2 Weaknesses

The background for this discussion includes UNIX-based interactive debuggers on various supercomputer systems incorporating the X interface as well as other debugging systems. In general, although IPD's commands are similar to UNIX interactive debugger commands, they lack the robustness associated with the UNIX commands. Also by comparison, IPD's minimal command repertoire limits its effectiveness. The greatest weakness is the lack of the X interface, which can save considerable session time. A subtle weakness is the fact that IPD possesses commands which have the functional equivalent of DBX debuggers, but have a different name. Aliases could be used to remedy this particular difference.

The following list represents notable deficiencies in comparison to Unix-based debuggers.

1. No conditional breakpoint capability.
2. No **when** command capability.
3. No **trace** command capability.
4. Difficult to save information from previous session to be used in restarting next session.
5. Can't output data/debug session information except to the log file.
6. Weak **display** command (vs. general print capability).

7. In order to address line numbers/variables not in the current scope the **procedure** and/or **file/procedure** must be present on the IPD command. DBX's **up/down** and **func/file** handle this situation efficiently.

An example of an X based interactive parallel debugging system is the Totalview Debugger within the Xtra Tool Set for the BBN TC2000 multiprocessing computer system. The obvious distinction between a graphical-oriented system such as Totalview and a line-oriented system, such as IPD is in the ability to present to the user in organized fashion significantly more meaningful information in a much shorter time frame. This directly impacts the users ability to more rapidly evaluate potential cause/effect relationships, especially complex, subtle or surprising interdependencies among processes (akin to iPSC/860 nodes).

Major features of Totalview include,

1. Multiple-window display
2. Window-per-process display of useful information (concept may not be feasible or even desirable much beyond medium-grained machines - such as the BBN's)
3. Pop-up menus eliminating command memorization
4. "Point and click" mechanism for breakpoint setting
5. Progressive "diving" into "objects" (routines,variable, etc.) for more detailed view
6. User selectable additional windows capability
7. Integrated, context-sensitive, on-line help facility
8. Spell correction on routine, variable names, etc.

The hierarchical window system available in Totalview begins with the "root window" which displays updated process state. "Panes" within an individual process window contain the stack trace, stack frame, and executing source code. For a given process window, users can select a wide variety of displayed objects such a routines, variables, breakpoints, stack frames, etc. to obtain progressively more detailed information. Separate windows are opened automatically and "clicked" closed. The user also has management over several windows (any process) devoted to displaying a specific type of information - such as all global variables, local variables, breakpoints, events (log), etc. The reference for the preceding information on Totalview was found in reference [3] (see section 5.6).

5.5 Known Bugs of IPD

The following command sequence causes a syntax error within IPD.

```
run;wait
```

5.6 References for IPD

[1] Intel iPSC/2 and Intel iPSC/860 Interactive Parallel Debugger Manual (I-9)

[2] Chapter 8 of the Intel Mini Manual (I-1)

[3] TC2000 Technical Product Summary, Revision 2.0, Nov. 1989 by BBN Advanced Computers, Inc.

6 MIMDizer

6.1 Overview of MIMDizer

MIMDizer is an interactive tool which assists the user in the creation, maintenance, and modification of computer programs for distributed memory parallel machines. It is an extension of FORGE developed by Applied Parallel Research Corporation and can be used to analyze and understand a large existing program through the FORGE capabilities of displaying global control flow, tracing global use of variables, and analyzing data dependencies among any parts of a program. For the transformation of a program to parallel, MIMDizer supplies a user interface for data decomposition of arrays and distribution of loop iterations to distributed memory processors. MIMDizer also provides the capability of defining a parallel program from scratch via interactive menus so that the user need not learn another parallel programming language. Complete documentation is available in references [1] and [2] (see section 6.6).

Transforming a serial FORTRAN program into a parallel version requires restructuring the program into a parallel form. The efficiency of the final parallel product is dependent on the programmer's understanding of the code and the problem being solved. MIMDizer is not a fully automated tool for generating a parallel code, but rather simplifies the task of porting an existing program to a distributed memory parallel machine.

Several steps are required to complete the transformation of a typical serial program to a parallel version. The first step involves determining which array or arrays are suitable for parallel processing. The FORGE part of MIMDizer is useful here for tracing the use of the array throughout the program and for determining if there are any data dependencies associated with the array. The second step will generally be to decompose the array across the processors. This step assigns "ownership" of portions of the array to particular processors and assures that the required portions of the array are available to those processors requiring them. The third step might be to distribute the loop iterations over multiple processors for the array being considered. It is primarily the responsibility of the user to avoid communication problems associated with distributing loop iterations in a manner inconsistent with the data decomposition. The end result of these three steps, or more likely several iterations of these steps, is the creation of a large database system comprised of four distinct parts: a symbolic database, a dataflow database, a parallel directives database, and a parallel model database. The fourth, and final, step in the transformation process is to invoke a precompiler which will use the contents of these

databases to instrument the original code into a paraliel version of the program.

6.2 Access to MIMDizer

MIMDizer is executed by entering the pathname at the shell command prompt. The pathname is

```
/ump/ipsc/home/psr/psr/distrib/forge
```

MIMDizer interfaces with the X Window System, so you will need to have your DISPLAY environment variable set for your display. The precompiler is invoked by entering the pathname

```
/ump/ipsc/home/psr/psr/distrib/pref77
```

with options to be discussed below.

6.3 Detailed Description of MIMDizer

MIMDizer is menu-driven and it is straightforward to learn your way around the system. Upon invocation, MIMDizer displays several menu entries in the right portion of the window. To get started, you have to create or select a serial package containing the original program. If this is your first session, there is of course no package to select, so you must create one. The most efficient way to select menu entries is to use the mouse to "pick" them by clicking the left button. To create the serial package the following steps are necessary:

```
Pick "Create a Package"
Enter the package name
Pick "Define Package"
Pick "Select Hardware File"
Pick the appropriate hardware file (e.g., "iPSC 860")
Pick "Add Source Files to Package"
Mark the desired source file(s) by picking them and then pick "Add"
Pick "RETURN" twice to return to the original menu
```

You have now created a package and could quit by picking "Exit". For your next session you would be able to "Select" this package or create a new one. From this point, the following steps illustrate a possible parallelization:

```
Pick "Analyze Program" Pick "Create" (in the lower window)
Pick the main program unit in the call chain in the right window
Pick "Parallelize for Distributed Memory"
```

```
Pick "Data Decomposition"
Pick "Decompose"
Pick "Create"
Pick "Type" to toggle between "CYCLIC", "BLOCK", or "REPLICATE"
Pick "Dimension" to toggle between 1 and 7
Pick "Allocation" to toggle between "FULL SIZE" and "SHRUNK"
```

The decomposition and allocation choices are straightforward. Cyclic decomposition is like dealing cards. The first element goes to processor one, the second to processor two, and so on. Block decomposition is like cutting the deck. The first set of elements goes to processor one, the second set to processor two, and so on. Replicate decomposition gives all the elements to each processor. Full size allocation stores the entire array on each processor. Shrunk allocation stores only those array elements required including any communicated values. Not all arrays can be shrunk. For example, an array initialized by a FORTRAN READ statement cannot be shrunk. Continuing with the parallelization process:

```
Pick "Save"
Pick the array(s) you want decomposed
Pick the decomposition type you defined
Pick "Apply"Pick "RETURN" twice
Pick "Analyze Automatically"
Pick the loops you want distributed
```

If you decompose on the first dimension of an array, you should also distribute on the loop corresponding to the first dimension. If you decomposed on the second dimension, distribute on the second dimension. MIMDizer will not tell you if you are being inconsistent here when using "Analyze Automatically". If you use "Analyze Interactively", MIMDizer will give some clues that you are inconsistent, but you have to look for them. Following the line "Preloop communication of", MIMDizer prints the array elements being communicated. If you are inconsistent the communication consists of one word at a time. If you see the message

*** The array *arrayname* needs cross iteration communication

in the lower window, you are probably doing something wrong and the resulting code will be extremely inefficient. Getting back to our example:

```
Pick "Save"
Pick "MENU"
Pick "Exit"
```

You have now decomposed and distributed your serial program. To precompile, invoke *pref77* with the following options

```
pref77 serialcode -p packname -o parallelcode
```

where **serialcode** is the name of your original source file
packname is the name of your MIMDizer package
parallelcode is the name of the output parallel source file

To compile the parallel code enter

```
if77 -o outf -node parallelcode -L/ump/ipsc/home/psr/psr/distrib/lib.860 -l libdd_n.a
```

6.4 Observations on Using MIMDizer

MIMDizer has the potential to be a valuable tool for users who have existing serial programs (with which they are familiar) which are to be transferred to parallel versions. Since MIMDizer is menu-driven, it is relatively easy to learn your way around the system. Several simple examples have been used to demonstrate that a user who is familiar with his program is capable of creating a parallel version with nearly linear speedup, even though he knows absolutely nothing about the node-to-node communication routines on the iPSC/860. The only weaknesses observed for MIMDizer concern diagnostics. For the novice user, especially, MIMDizer would be improved by issuing appropriate and meaningful warning messages when things are being done which obviously have a negative impact on the efficiency of the parallel program. In the absence of such diagnostics, the user is forewarned to pay particular attention to the decomposition and distribution processes. It is up to the user to assure that there are no dependencies in the data being decomposed and to assure that the decomposition and distribution are performed in a compatible manner.

6.5 Known Bugs of MIMDizer

Although invoking the Parallel Runtime Monitor, as described in [2], led to the creation of several "diag" files, attempts to invoke the "Node Profiler" option were unsuccessful.

6.6 References for MIMDizer

[1] FORGE User's Guide, Volume III
[2] MIMDizer User's Guide, Version 7.10, June 1991

7 Parallel Performance Analysis Tools (PAT)

7.1 Overview of PAT

The iPSC/860 Parallel Performance Analysis Tools (PAT) is a set of utilities for gathering statistics on and analyzing program execution, communication performance, and event traced performance of application programs on the iPSC/860 system. The version described in this document is the Intel Supercomputer Systems Division version of a performance analysis product developed by the ParaSoft Corporation.

The PAT utilities gather performance data at runtime and output the data to disk when the application completes. The PAT profiling utilities support both the FORTRAN and C programming languages. PAT provides a set of analysis tools that convert the performance data to tabular and graphical forms that can be analyzed interactively. The graphical forms can be viewed with SunView or X Window System (X). The PAT utilities provide hardcopy output in PostScript form. Information concerning PAT include [1] for detailed information, the man page *pat* for an on-line summary, and the file

```
/ump/ipsc/share/local/PAT-ipsc/README/larc
```

for access and other general information. Questions about PAT may be directed via e-mail to:

```
mpp@fiddler.larc.nasa.gov
```

7.2 Access to PAT

The iPSC/860 Parallel Performance Analysis Tools (PAT) analyzes performance of application programs on the iPSC/860 system, and is used in conjunction with the iPSC/860 C and FORTRAN compilers. Thus, code profiling can be instrumented on platforms having access to the iPSC/860 compilers. The analyzing utilities are available on systems holding the host software (including Fiddler), and can be invoked from a graphical device (SunView or X) or from a non-graphical terminal (vt100).

The software package Express also developed by ParaSoft Corporation, see Chapter 2, contains many of the capabilities, program commands, and system calls as PAT. On systems where both software packages are available, the use of the environment variables (for *path* and *man* pages)

determine which package is being invoked. The default *path* and *man* pages provided by the system administration default to the PAT utility, not Express. The PAT *man* pages will have the phrase *i*PSC(Reg.)860 PAT Utilities in the heading. Chapter 2 describes how to change the environment variables to use Express.

7.3 Detailed Description of PAT

The PAT performance data is collected automatically by using compiler switches and environment variables or selectively through the use of PAT C or FORTRAN calls that are compiled with the application program. The iPSC/860 system provides three methods to insert and control the use of PAT system calls in the application code:

* Compiler switches to automatically profile/trace application performance

* Environment variables that interactively toggle the gathering of performance data

* A programmatic interface that allows the manual insertion of PAT system calls at user specified points in the application code

The recommended method of gathering profiling data uses switches of the C and FORTRAN compilers to turn one or more of the profilers on or off for the code being compiled. This method automatically inserts the appropriate PAT procedures and are at the start and exit of every procedure/subroutine.

The PAT utilities provide separate post mortem tools to analyze the major areas of parallel program performance on the iPSC/860 system. The execution analyzer tool, *xtool*, displays the time spent in individual routines. The communication analyzer tool, *ctool*, shows the time spent in communication and I/O. The event analyzer tool, *etool*, presents the interactions among processors and monitors user-specified events. These analysis tools convert the gathered performance data into tabular and graphical forms that can be analyzed interactively.

7.3.1 Profiling Methods

The PAT performance data is collected through the use of PAT C or FORTRAN system calls that are compiled with the application program. The system calls can be placed in the application code automatically, or they can be added manually. The following sections provide an overview of the three profiling utilities. Section 7.3.3 describes the compiler switches to instrument manual and automatic data gathering.

7.3.1.1 Profiling the Program Execution

The Execution Profiler provides two distinct statistical methods. The first method, the sampling method, regularly polls the processor execution state. At a specified time cycle (10 milliseconds) a system routine looks at the current instruction being executed in the user application and increments a counter noting the memory address. Based on the sampling information *xtool* builds a histogram of the frequencies of hits in various program areas. Then based on the histogram PAT determines the amount of time spent in particular routines.

The second method, the counting method, counts the number of times the profiled routines are executed. When execution profiling is selected procedure calls are inserted by the compiler at the start and end of each compiled routine.

Notes: The core of the profiling methods are based on the UNIX utility profil(), the sampling technique is not foolproof (e.g., when the sample rate is same as event rate, then a false profile will result) and *xtool* often requires a lot of memory to make a histogram.

The following skeleton code, described in reference [1] illustrates the manual control of the execution profiler.

```
      PROGRAM XPRTST
      INTEGER PRFBUF(2048), PRFSCL
C
C-- This value is 0x2000 (hexadecimal)
      PARAMETER (PRFSCL = 8192)
C
C-- This is the name of a function in the program, low
C-- in memory. A suitable candidate can usually be found
C-- by looking through the "linker map".
      EXTERNAL F_MAIN
C
C-- Start up profiler if user selected -mx option.
C
      ISTAT = KXPINQ()
      IF(ISTAT .NE. 0) THEN
            CALL KXPINI(PRFBUF, 8192, F_MAIN, PRFSCL)
            CALL KXPON
      ENDIF
C
C-- Execute application code with profiler running.
C
      ...
C
C-- Program over, dump data and exit.
C
      IF(ISTAT .NE. 0) CALL KXPDMP('exprof.prx')
      STOP
      END
```

7.3.1.2 Profiling the Program Communication

On each node, data is accumulated to measure:

* Time spent calculating, communicating between processors, and performing I/O functions. This provides an estimate of program overheads and efficiency.

* Total number of calls to the communication system. This provides a simple estimate of load imbalances.

On each node, the following function information is recorded:

* Number of calls to each individual function.

* Distribution of return values from each function (i.e., message length written, message length read, number of objects broadcast).

The following skeleton code, described in reference [1] illustrates the manual control of the communication profiler in an application program. The only case in which explicit calls are needed is when more careful control is required over the profiler and the data it writes to disk.

```
      PROGRAM CBXPRF
C
C-- Start off profiler.
C
      ISTAT = KCPINQ()
      IF(ISTAT .NE. 0) CALL KCPON
C
      ...
C
C-- Program over, Dump data again and exit.
C
      IF(ISTAT .NE. 0)CALL KCPDMP('exprof.prc')

      STOP
      END
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

7.3.1.3 Profiling the Program Events

The event profiler records "events" in an internal log for later analysis. An event is a user-specified point in the execution. The following information is recorded every time an event occurs.

Event information:

* a time-stamp at which the event occurred
* an index value indicating the nature of the event
* a program variable at the time of the event

Optional information:

* a title which identifies an event based on a given index value
* a printf-style format string command to control the printing of a stored program variable

The following skeleton code, described in reference [1] illustrates the manual control of the event profiler.

```
      PROGRAM EPRTST
      INTEGER LOGBUF(2048), LABBUF(256)
      REAL ENERGY, RESID, GRIND, CRUNCH
      INTEGER ITER, I
```

```
C-- Setup profiler and make labels for indices.
C--  If asked to do so at runtime start the thing up.
     CALL KEPINI(LABBUF, 1024, LOGBUF, 8192)
     CALL KEPLAB(1, 'Outer loop', 'Iteration %d')
     CALL KEPLAB(2, 'After crunch', 'Energy = %d')
     CALL KEPLAB(3, 'Inner loop', 'resid = %d')
     ISTAT = KEPINQ()
     IF(ISTAT .NE. 0) CALL KEPON
C--Start application code, then go into main loop.
     ...
     DO 10 ITER=1,100
          CALL KEPADD(1, ITER)
          ENERGY = CRUNCH(ITER)
          CALL KEPADD(2, INT(ENERGY))
          DO 20 I=1,4
               RESID = GRIND(ENERGY)
               CALL KEPADD(3, RESID)
     20   CONTINUE
     10   CONTINUE
C
C-- Program over; dump data to host for later analysis.
     CALL KEPDMP('exprof.prc')
     STOP
END
```

Notice that the **KEPADD()** and **KEPLAB()** calls are completely safe even if **KEPINQ()** returns 0 and the profiler is not enabled.

7.3.2 Analyzing Methods

The PAT utilities provide separate post mortem tools to analyze the major areas of parallel computer performance on the iPSC/860 system. The execution analyzer tool, *xtool*, displays the time spent in individual routines. The communication analyzer tool, *ctool*, shows the time spent in communication and I/O. The event analyzer tool, *etool*, presents the interactions among processors and monitors user-specified events. These analysis tools convert the gathered performance data to tabular and graphical forms that can be analyzed interactively.

The synopsis for the use of each tool is:

xtool [-H...] [-I] [-I directory] [-M menustyle] [-n PSname] [-p] [-t topno] [-T...]
program_name [log_file_name]

ctool [-b nbins] [-H...] [-p] [-T...] [-M menustyle] [-L DBname] [-m PSname]
program_name [log_file_name]

etool [-e] [-H...] [-L DBname] [-M menustyle] [-n PSname] [-p] [-t] [-T...]
program_name [log_file_name]

The following is a partial list of options from the PAT *man* page. Please refer to the *man* pages or reference manual for a complete description.

-H...	Specify the type of output device required for "Hardcopy" options. By default hardcopy output is produced in PostScript form. The interpretation of this switch is similar to that of the *-T* option.
-I	After processing other command line options enter "interactive" mode as shown below. This switch is only active when used in conjunction with the *-p* switch.
-I *directory*	Causes the named directory to be searched when looking for the source code to the various routines in the program. By default only the current directory is searched.
-M *menustyle*	Use an alternative menu style. By default the tool attempts to use "three-dimensional" menus that may not function correctly on certain monitors. Substituting alternate numeric values in this switch uses other menu types of which *-M0* should be viable on any kind of monitor.
-n *PSname*	The base name given to the hardcopy output files created by the system. The actual filename used is created by appending a numeric value and the string *.ps* to the name given to this switch.
-p	Suppress graphical output. The analysis results are presented in tabular form on stream stdout.
-t *topno*	Instructs xtool to print data from the topno most active routines. This switch is only active when used in conjunction with the *-p* switch.
-T...	Use an alternative graphical device for output. The supported graphical devices and their abbreviations can be found in the "Introductory Guide" for your system.

7.3.2.1 Analyzing Execution Profile (xtool)

The *xtool* command is available at the system prompt on the host processor, and is used to examine and analyze the log file created with the execution profiler. The only required argument is the name of the executable program to be analyzed. Another argument is the name of the file containing the profiled data. This argument is optional and if omitted defaults to *exprof.prx*. The execution profiler uses the data contained in a symbol table generated by the execution of the program compiled with the *-Mperf* compiler switch.

The *xtool* command presents a separate table on *stdout* from each node. The information contained in each table is:

* An identifier indicates which node the data is from.

* A summary of the busy and idle time in each processor. The term "idle time" denotes when the CPU is not actively executing the process (e.g., waiting for a message to arrive). All other classes of activity are counted as "busy".

* A count of the number of profiling "misses". Since the buffer supplied to the profiling function profil may not be fine enough to map the whole program, the execution profiler will "miss" occasionally, that is the program will execute at an address outside the region mapped by the profil call. In this case the "miss" counter is incremented. The ratio of hits to misses provides an indication of the effectiveness of the profile obtained.

* A list of the 20 most heavily used functions in the program is provided. Associated with each is the fraction of the total profiling events.

If the *xtool* command is invoked without the *-p* switch then a menu-driven, graphical interface is provided. A full list of the available options is presented in the PAT manual.

By default the graphical system used is either a Sun-3 or a Sun-4 workstation, along with a PostScript printer for hardcopy output. The *-T* switch can be used to redirect output to another device. Similarly the *-H* switch is used to redirect the hardcopy output.

EXAMPLES:

To examine the execution profile data in a file called *test1.prof* created by the program myprog execute the command:

```
xtool myprog test1.prof
```

To analyze the default profile data file from the program named **myprog** using simple menus under X, execute the command:

```
xtool -TX -M0 myprog
```

An "interactive" mode may be used in place of the graphical mode. To enter the interactive mode use:

```
xtool -p -I myprog
```

After completing normal *-p* processing you see the prompt

```
xtool>
```

At this prompt the following commands may be executed:

help	Prints a summary of available commands.
top #	Prints the normal tabular output showing only the indicated number of routines for each node.
quit	Terminate xtool and return to the shell.
node #	Restrict output to the particular node indicated. Use the value "-1" to indicate output from all nodes.
output file	Redirect printed output to the named file. Using the command with no file name redirects output to the terminal.

7.3.2.2 Analyzing Communication Profile (ctool)

The *ctool* command is available at the system prompt on the host processor, and is used to examine and analyze the log file created with the communication profiler. The only required argument is the name of the executable program to be analyzed. Another argument is the name of the file containing the profiled data. This argument is optional and if omitted defaults to *exprof.prv*.

If the *-p* switch is given, this command presents a separate table on stdout from each node. The information contained in each table is:

* An identifier indicating which node the data is from.

* A summary of the calculation, communication, and I/O times in the processor. "Communication" time consists of all inter-node and basic host-node communication, "I/O" time consists of all I/O requests (i.e., calls to read, write, fopen, etc.).

* A summary of the time spent in, number of calls, and errors incurred in each communication function called by the processor. This information provides an overview of the total communication pattern. The "error" count is also a good place to look for obscure bugs.

* A breakdown of the values returned by the communication functions. The return values are binned logarithmically - the column headed "8" indicates the frequency of return values in the range 8 thru 15 (inclusive).

The communication profiler can be useful in the detection of programs sending too much data in their messages. These will show up in the histogram output. This data appears on *stdout.*

If the ctool command is invoked without the *-p* switch then a menu-driven, graphical interface is provided. A full list of the available options is presented in the PAT manual.

By default the graphical system used is either a Sun-3 or a Sun-4 workstation, along with a PostScript printer for hardcopy output. The *-T* switch can be used to redirect output to another device. Similarly the *-H* switch is used to redirect the hardcopy output.

EXAMPLES:

To examine the communication profile data in a file called *test1.prof* created by the program named *myprog,* execute the following command:

```
ctool myprog test1.prof
```

To analyze the default profile data file from the program named *myprog* using simple menus under X execute the following command:

```
ctool -TX -M0 myprog
```

ERRORS

The *ctool* utility uses an internal data-base to find the names and identifiers of the functions being profiled. If this file is missing you see the error "No system call data-base". This normally means that your system is not installed properly. Either contact your system administrator or use the *-L* switch to specify an alternate path.

If the file is present but garbled or otherwise incorrect *ctool* appears to function correctly but either gives the wrong names to subroutines or does not list anything at all. In this case the file should be recreated using the correct system call information.

7.3.2.3 Analyzing Event Profile (etool)

The *etool* command is available at the system prompt on the host processor, and is used to examine and analyze the event log created with the event profiler. The only required argument is the name of the executable program to be analyzed. Another argument is the name of the file containing the profiled data. This argument is optional and if omitted defaults to *exprof.pre.*

If the *etool* command is invoked without the *-p* switch then a menu-driven, graphical interface is provided. A full list of the avallable options is presented in the PAT manual.

By default the graphical system used either a Sun-3 or a Sun-4 workstation, along with a Postscript printer for hardcopy output. The *-T* switch can be used to redirect output to the another device. Similarly the *-H* switch is used to redirect the hardcopy output.

EXAMPLES:

To examine the event profile data in a file called *test1.prof* created by executing the program myprog, execute the command:

```
etool myprog test1.prof
```

To analyze the default profile data file from the program name *myprog* using simple menus under X, execute the command:

```
etool -TX -M0 myprog
```

If only the toggle information is required from the profile data use a command similar to:

```
etool -p -t myprog toggle.pre
```

ERRORS

The *etool* utility uses an internal data-base to find the names and identifiers of the functions being profiled. If this file is missing you see the error "No system call data-base". This normally means that your system is not installed properly. Either contact your system administrator or use the *-L* switch to specify an alternate path.

If the file is present but garbled or otherwise incorrect, *etool* appears to function correctly but either gives the wrong names to subroutines or does not list anything at all. In this case the file should be recreated using the correct system call information.

7.3.3 Compilation and Execution

The PAT performance data is collected through the use of PAT C or FORTRAN system calls that are compiled with the application program. The system calls can be placed in the application code automatically, or they can be added manually.

The iPSC system uses the following methods to insert PAT system calls in the application code:

* Compiler switches to automatically profile/trace application performance
* Environment variables that interactively turn on/off the gathering of performance data
* A programmatic interface that allows the manual insertion of PAT system calls at specified points in the application code

The *-Mperf* switch is available with the iPSC/860 C and FORTRAN compilers, and turns on PAT instrumentation code. The syntax for the use of this switch is

-Mperf [= {prof | comm | event} [= {auto | manual}] [,...] | {auto | manual}]

prof Enables the PAT execution profiling.

comm Enables the PAT communication tracing.

event Enables the PAT event tracing.

auto Specifies that performance monitoring code is to be invoked automatically.

manual Specifies that performance monitoring code is to be called manually using PAT programmatic procedure calls.

EXAMPLES:

```
        /* turn on all PAT instrumentation        */
if77 -Mperf -o ftest *.o -node

        /* turn on automatic execution profiling */
if77 -Mperf=prof=auto *.f -o ftest -node

        /* turn on comm. and event manually    */
icc -Mperf=comm=manual,event=manual
```

The environment variable, **EXPROF_SWITCHES** allows you to set or reset any or all the PAT profile/trace tools. The syntax for the **EXPROF_SWITCHES** environment variable is:

setenv EXPROF_SWITCHES ["[c][e][x]"]

where the option *c* represents communication tracing, *e* represents event tracing, and *x* represents execution profiling.

The environment variable, **EXPROF_OPTS** allows you to set the maximum number of log and label entries allowed in the log file used by the etool utility. The syntax for the **EXPROF_OPTS** environment variable is:

setenv EXPROF_OPTS "{ elogs | elabs } : new_value [; ...]"

where the *elogs* represents the log entries, and *elabs* represents the label entries, and new_value represents the new value used.

EXAMPLES:

```
/* all PAT instrumentation is to profile data*/
setenv EXPROF_SWITCHES "cex"
 /* set limit of logfile and labelfile */
setenv EXPROF_OPTS "elogs:1500;elabs:1024"
```

The following is a simple program to demonstrate the compilation and execution process.

```
      program f2
      integer iter
      pid = mypid()
      mynod = mynode()
      call keplab(1,'outer loop','iteration %d')
      do 1 iter=1,3
          call kepadd(1,iter)
1         write(6,*)"hello world from  node",mynod," process id ",pid
      end
```

The following demonstrates the compilation and execution process. Note when used in conjunction with PAT, the *waitcube* command ensures that data collection has completed before the system prompt is returned. The user can then safely release the allocated cube. *cubeexec* is the preferred command to accomplish this automatically.

```
fiddler% setenv EXPROF_SWITCHES "cex"
fiddler% if77 -Mperf -o f2 f2.f -node
fiddler% cubeinfo
(host) cubeinfo: There is no attached cube
fiddler% getcube -t4
getcube successful: cube type 4m8rxn4 allocated
fiddler% load f2; waitcube
 hello world from node        3 process id  0.0000000
 hello world from node        1 process id  0.0000000
 hello world from node        0 process id  0.0000000
 hello world from node        2 process id  0.0000000
 hello world from node        3 process id  0.0000000
 hello world from node        1 process id  0.0000000
 hello world from node        0 process id  0.0000000
 hello world from node        2 process id  0.0000000
 hello world from node        3 process id  0.0000000
 hello world from node        1 process id  0.0000000
 hello world from node        0 process id  0.0000000
 hello world from node        2 process id  0.0000000
Dumping to "exprof.prc"
Dumping to "exprof.pre"
Dumping to "exprof.prx"
fiddler% relcube
relcube released 1 cube
fiddler%
fiddler% etool -p f2
Etool Version 3.1.2 -- Copyright (C) 1991 ParaSoft
Reading symbol table "f2":  100%
```

```
Symbol table: 728 public, 30 local.
Index look-up failed for symbol 372

Node  0
==========

  No toggle data.

 1. outer loop
T = 6.154 (ms)  Begin _MAIN_.
T = 6.797 (ms)  Index =      1  "Iteration 0".
T = 8.035 (ms)  Begin _write, fildes=1 nbyte=62.
T = 8.174 (ms)  Begin __cwrite, fildes=1 len=62.
T = 8.945 (ms)  Begin __masktrap, state=1.
T = 9.032 (ms)  End   __masktrap, returned=0.
T = 9.579 (ms)  Begin __csendrecv, type=1000000020 tonode=4.
T = 9.666 (ms)  Begin __flick.
T = 2050.084 (ms)  End   __flick.
T = 2050.108 (ms)  End   __csendrecv, typesel=2000000021.
T = 2050.142 (ms)  Begin __masktrap, state=0.
T = 2050.157 (ms)  End   __masktrap, returned=1.
T = 2050.182 (ms)  Begin __masktrap, state=1.
T = 2050.193 (ms)  End   __masktrap, returned=0.
T = 2050.460 (ms)  Begin __masktrap, state=0.
T = 2050.474 (ms)  End   __masktrap, returned=1.
T = 2050.620 (ms)  End   __cwrite, returned=62.
T = 2050.632 (ms)  End   _write, returned=62.
T = 2050.731 (ms)  Index =      1  "Iteration 1".
T = 2051.120 (ms)  Begin _write, fildes=1 nbyte=62.
T = 2051.149 (ms)  Begin __cwrite, fildes=1 len=62.
T = 2051.215 (ms)  Begin __masktrap, state=1.
```

7.3.4 System Calls

The following is a list of PAT system calls. More detailed information is available in the *man* pages for the following PAT commands, and in reference [1].

CPROF	Control communication profiler (C language system call). Also referenced by: CPROF_OFF CPROF_ON
CPROF_DMP	Dump communication profile data to disk from the application program (C language system call).
CPROF_INQ	Determine runtime status of profiling system (C language system call).
CTOOL	Analyze Communication Profile (Command).
EPROF	Event driven profiler (C language system call). Also referenced by: EPROF_ADD EPROF_INIT EPROF_LABEL EPROF_OFF EPROF_ON

EPROF_DMP	Dump event profile data to disk from an application program (C language system call).
EPROF_INQ	Determine runtime state of the event profiling system (C language system call).
EPROF_TOGGLE	Statistical analysis of code sections (C language system call Also referenced by: EPROF_TOGINIT ETOGGLE
ETOOL	Analyze Event Profile (Command).
KCPDMP	Write communication profile data to a file (FORTRAN language system call).
KCPINQ	Determine runtime status of execution profiler (FORTRAN language system call).
KCPROF	Control communication profiler (FORTRAN language system call). Also referenced by: KCPOFF KCPON
KEPROF	Event driven profiler (FORTRAN language system call). Also referenced by: KEP KEPADD KEPINI KEPLAB KEPOFF KEPON
KEPDMP	Write event profile data to a file (FORTRAN language system call).
KEPINQ	Determine runtime status of the event profiling system (FORTRAN language system call).
KEPTOG	Calculate program statistics (FORTRAN language system call). Also referenced by: KEPTGI
KXPDMP	Write execution profile data to a file (FORTRAN language system call).
KXPINI	Low level execution profiler (FORTRANlanguage system call).
KXPINQ	Determine runtime status of the execution profiler (FORTRAN language system call).
KXPROF	Control execution profiler (FORTRAN language system call). Also referenced by: KXPOFF KXPON
XPROF	Control execution profiler (C language system call). Also referenced by: PROF_OFF XPROF_ON
XPROF_DMP	Dump execution profile data to disk (C language system call).
XPROF_INIT	Low level execution profiler (C language system call).
XPROF_INQ	Determine runtime status of execution profiler (C language system call).
XTOOL	Analyze Execution Profile (Command).

7.4 Observations on Using PAT

Section 7 has provided an overview of the iPSC/860 Parallel Performance Analysis Tools (PAT), a detailed description of its functionality and use, and a summary of the commands. The following comments are based on a preliminary use of PAT.

The general performance information is easy to obtain, usually through the automatic profiling method. However, this approach can yield large amounts of information. Thus manual instrumentation is often necessary to reduce the amount of performance information and isolate performance information down to critical sections of code.

The default use of PAT assumes that the same executable is running in every node. PAT can be used on applications where different executables are running on different nodes by using manual instrumentation. That is, run PAT for executable A, look at the data. Then run PAT on executable B, etc.

Preliminary evaluation indicates that PAT overhead averages 10%, but varies depending on the computation and communication make-up of the application.

When used in conjunction with PAT, the *waitcube* command ensures that data collection has completed before the system prompt is returned. The user can then safely release the allocated cube. Otherwise the user may release the cube prior to all data being collected. When this occurs, no error message will be given, but subsequent data analysis will either not be allowed or yield incomplete results. Use of *cubeexec* avoids this type of problem.

7.5 Known Bugs of PAT

No bugs are known within PAT at this time.

7.6 References for PAT

[1] iPSC/860 Parallel Performance Analysis Tools Manual (I-10)

8 Parallel Virtual Machine (PVM)

8.1 Overview of PVM

Parallel Virtual Machine (PVM) is a free software package that is being developed at Oak Ridge National Laboratories (ORNL). PVM allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. It is a library (two libraries if you use FORTRAN) and a daemon process. The purpose is to couple several resources in a parallel fashion to use the best properties of a particular machine for an application with moderately large granularity. For example, a CRAY Y-MP could be used to perform vector calculations while a iPSC/860 could do the highly parallel calculations.

The PVM daemon communicates to other machines through sockets, a software mechanism. Communication between PVM processes and the iPSC/860 takes place through the System Resource Manager (SRM). Since this can lead to very complicated communication programming, it is recommended to use PVM on the iPSC/860 only for applications with very large levels of granularity. A current research effort is porting the PVM libraries to the individual iPSC/860 nodes.

As of today, PVM only parallelizes a network of computers and does NOT parallelize locally. Therefore, on the cube, the regular Intel communication libraries (csend, gdsum, etc.) are used for inter processor communication. The nodes send messages to the host program, and then the host program sends the messages to the other PVM processes and vice versa.

PVM for the iPSC/860 has some special considerations, therefore this report will discuss creating and using PVM on the iPSC/860. The PVM users' guide contains information for general installation and use.

8.1.1 OS and Platforms

PVM version 2.4.1 was developed for several platforms including:

ARCH	Machine
C2MP	Convex
CM5	Thinking Machines CM-5
CRAY	Cray (UNICOS 6 or greater)
HP9K	HP 9000 (Snake)
KSR1	Kendall Square
i860	Intel RX Hypercube
PMAX	Dec/Mips arch (3100, 5000, etc)
RIOS	IBM/RS6000
SGI	Silicon Graphics IRIS
SUN3	Sun 3 (SunOS 4 or greater)
SUN4	Sun 4, 4c, Sparc, etc

8.2 Access to PVM

If PVM is not installed on your system, you may install PVM yourself. You can compile the PVM daemon and library in your own directory. To install PVM, you must first get some files from mass storage. If you get the message "Permission Denied," then make sure your *.rhosts* file contains an entry for **Sabre**. See section 5.5 of SNS Programming Environment (A-8) for details.

```
masget @trey/Parallel/pvm2.4.1.shar
masget @trey/Parallel/pvm.ps
masget @trey/Parallel/pvm_refcard.ps
masget @trey/Parallel/make_pvm
masget @trey/Parallel/README.pvm
```

You must unpack *pvm2.4.1.shar* in your home directory. To unpack type

```
sh pvm2.4.1.shar
```

You should now have the directory *pvm2.4*. So that the PVM daemon can find your executables, make a symbolic link by typing

```
ln -s pvm2.4 pvm
```

You should now have the link *pvm@*. Now you need to create the PVM libraries and PVM daemon by executing the *make_pvm* script. If you have problems with the script, send e-mail to:

```
mpp@fiddler.larc.nasa.gov.
```

8.3 Detailed Description of PVM

8.3.1 Compiling with PVM

For the iPSC/860, the PVM daemon is built and run on the SRM, therefore the host program acts as the node for the PVM network. Since the PVM library is built on the SRM and not the iPSC/860 nodes, you must use Intel's communication library for inter processor communication. The host program acts as an interpreter between the cube and the PVM network.

Here is an example PVM host file in FORTRAN for the iPSC/860:

```
c
c This host program enrolls itself as PVM node process.  Then,
c it gets an 8 node cube and loads the intel_node program onto
c the cube.  The host receives a message from the cube and
c then that message is then broadcast to the other PVM processes.
c
      program pvm_host
      implicit real*8(a-h,o-z)
      parameter(nodes = 8)
      character*4 sznode

c enroll the pvm host program
      call fenroll("pvm_host\0",kn)
      if(kn .lt. 0) then
        write(*,*) 'I could not enroll Intel version host'
        stop
      end if

c get the cube, set the pid and load the node program
      write(sznode,'(i4)') nodes
      call getcube('pvm_cube',sznode,'',0,'')
      call setpid(npid)
      call load('/ump/ipsc/home/larc/trey/pvm/i860/intel_node',-1,npid)

c get the value of tm from the cube
      call crecv(100,tm,8)

c now that I have the message from the cube, send
c it out to the other PVM processes using PVM message passing
      call finitsend()
      call fputndfloat(tm,1,istat)
      call fsnd("pvm_node\0",-1,200,istat)

c kill the cube and release it
      call killcube(-1,0)
      call relcube('pvm_cube')

c quit PVM
      call fleave()
      stop
      end
```

The following is a non-cube PVM host program example. Note that this host program only needs to initiate the node programs to execute on the PVM.

```
c
c  This host program enrolls itself, then initiates the other node processes.
c
      program pvm_host
      implicit real*8(a-h,o-z)

c  enroll the pvm host program
      call fenroll("pvm_host\0",ihost)
      if(ihost .lt. 0) then
        write(*,*) 'I could not enroll host program'
        stop
      end if

c  see how many machines are configured in the PVM
      call fpstatus(nproc, nformat, info)

c  initiate the PVM nodes processes.  Since arch is equal to NULL, PVM will
c  decide which machine to initiate the node process "pvm_node\0" on.
      arch = "\0"
      do 100 i= 0, nproc - 1
        inst = i
        call finitiate("pvm_node\0",arch,inst)
        if(inst .lt. 0) then
          write(*,*) 'failed to initiate at process ',inst
          stop
        end if
100   continue

c  quit PVM
      call fleave()
      stop
      end
```

A sample makefile for a PVM host (and node program) on the SRM of the iPSC/860 (f77 is the SRM compiler) is:

```
####### Start of makefile for Intel PVM using SRM host ######

ARCH = I860

FLAGS =     -O3 -Mvect=unroll
NFLAG =     -i860 -node
HFLAG =     $(HOME)/pvm/f2c/$(ARCH)/libf2c.a \
            $(HOME)/pvm/src/$(ARCH)/libpvm.a -lrpc -lsocket -host

all: host node

host:
      f77 -o host host.f $(HFLAG)
```

```
        mv host $(HOME)/pvm/$(ARCH)

node:
        if77 $(FLAGS) -o node node.f $(NFLAG)
        mv node $(HOME)/pvm/$(ARCH)

clean:
        /bin/rm -f host node core

######end of makefile ##############
```

8.3.2 Running a PVM Program

To start the PVM, the PVM daemons must be executed on each machine. The command to start all of the PVM daemons on each machine is:

```
~/pvm/src/$ARCH/pvmd [-i] hostfile
```

where *hostfile* is a file containing a list of the machines in the PVM. The *-i* is the only option for the pvmd command. It requests that PVM start the daemon process interactively. If the *-i* option is used, a command editor is run allowing the user to query PVM on the status of processes and the present configuration of PVM.

The commands PVM recognizes in interactive mode are:

barr	show barriers
conf	shows the current configuration of machines in PVM
help	show all commands available and gives description of each
kill	kill pvm processes
ps	show process status
quit	kill all pvmd processes and exit PVM
reset	kills currently running PVM processes

In interactive mode, the user can terminate all PVM processes by typing quit at the **pvm** prompt.

A sample host file looks like:

```
# this is a comment (a # in the first column)
# this is a sample hostfile
yoursun.larc.nasa.gov
bluecrab.larc.nasa.gov lo=login_id pw dx=~/pvm/src/I860/pvmd
#end of hostfile
```

You can specify three options in the host file. The *lo* option tells PVM what login name to use for that machine. The *pw* option requests your password for that machine in the event that a *.rhosts* is not available. The

dx option allows the user to specify where the PVM daemon is located. The default location is

```
/tmp/pvm/pmvd
```

If the PVM daemons are successful in starting, the message "pvm is ready" appears. If the PVM returns a message "pvmd garbled response" or "expected pvmd but got "", this usually means that PVM can not read the **password** file. To remedy this situation, try using the pw option in the hostfile or checking your *.rhosts* file.

An example demonstrating how to compile and run PVM on the iPSC/860 is located on **Fiddler** in the directory

```
~trey/pvm/example
```

The file

```
~trey/pvm/example/README
```

contains instructions on how to compile and run the example.

8.3.3 Description of FORTRAN Subroutines in PVM

The following is a list of subroutines and a description of their use. The FORTRAN subroutines are wrappers for the C functions in PVM. The integer variable **info** returns a negative number if an error occurs.

Declarations

```
integer         cnt, info, inum, len, mt, ncpus, np(cnt), ntypes
integer         pstatus, type, types(ntypes)
integer*2       jp(cnt)
character       arch, cp, host, msg, name, proc
real            fp(cnt)
real*8          dp(cnt)
complex         xp(cnt)
complex*16      zp(cnt)
```

Initialization

Subroutine	Description
call fenroll(proc,inum)	enrolls process in PVM and returns inum, error if inum < 0.
call finitiate(proc,arch,inum)	initiates a new process on a specified architecture and returns instance number(inum). inum less than zero if error occurs. If arch is NULL, then PVM chooses an architecture.
call finitiatem(proc,host,inum)	initiates a new process on a specified machine (host) and returns instance number. inum less than zero if error occurs. If host = ".", then initiating machine is used.
call fwhoami(proc,inum,info)	returns component name (proc) and instance (inum).

Information

Subroutine	Description
call fpstatus(ncpus,ntypes,info)	returns number of hosts and data formats (ntypes).
call fstatus(proc,inum,pstatus)	pstatus = 1 if specified component is active, otherwise pstatus = 0.

Sending

Subroutine	Description
call finitsend()	initialize send buffer.
call fputbytes(cp,cnt,info)	puts number of bytes in cp(cnt) into buffer.
call fputncplx(xp,cnt,info)	put complex array into buffer.
call fputndcplx(zp,cnt,info)	put double complex array into buffer.
call fputndfloat(dp,cnt,info)	put double precision array into buffer.
call fputnfloat(fp,cnt,info)	put real array into buffer.
call fputnint(np,cnt,info)	put integer array into buffer.
call fputnlong(np,cnt,info)	put long integer array into buffer.
call fputnshort(jp,cnt,info)	put short integer array into buffer.

Subroutine	Description
call fputstring(cp,info)	put character string into buffer.
call fputstring(cp,len,info)	put character string into buffer.
call fsnd(proc,inum,type,info)	sends message in send buffer to the specified instance (inum) of component. If inum = -1. then broadcast to all instances.

Receiving

Subroutine	Description
call fgetbytes(cp,cnt,info)	returns number of bytes.
call fgetncplx(xp,cnt,info)	get complex array from buffer.
call fgetndcplx(zp,cnt,info)	get double complex array from buffer.
call fgetndfloat(dp,cnt,info)	get double precision array from buffer.
call fgetnfloat(fp,cnt,info)	get real array from buffer.
call fgetnint(np,cnt,info)	get integer array from buffer.
call fgetnlong(np,cnt,info)	get long integer array from buffer.
call fgetnshort(jp,cnt,info)	get short integer array from buffer.
call fgetstring(cp,info)	get string from buffer.
call fgetstring(cp,len,info)	get string from buffer.
call fprobe(type,mt)	probe for message arrival of specified type or 'any' if type= -1. Returns type or -1 (not arrived).
call fprobemulti(ntypes,type,mt)	same as probe, but permits specifying an array of ntypes message types.
call frcv(type,mt)	receives a message of specified type or 'any' if type = -1 (Blocking).
call frcvinfo(len,type,proc,inum,info)	returns the length, type and sender of last frcv or probe.
call frcvmulti(ntypes,types,mt)	same as rcv, but permits specifying an array of ntypes message types.

Synchronization

Subroutine	Description
call fbarrier(name,cnt,info)	blocks caller until cnt calls with same barrier name made. name can NOT be reused.
call fready(name,info)	sends signal with specified (abstract) name.
call fwaituntil(name,info)	suspends caller until specified signal name occurs.

Termination

Subroutine	Description
call fleave()	process existing PVM. This should be the last call to flush all PVM processes.
call fterminate(proc,inum,info)	terminates a specified component.

8.3.4 Global Subroutines

Many global routines are absent from the standard PVM system, therefore several locally developed routines have been written. These routines are written to provide the user with similar global routines provided by Intel (**gdsum**, **gdhigh**, etc). These global routines emulate Intel's global routines in table A-7 of reference [1]. This library of routines is located on mass storage. To get the file, type

```
masget @trey/Parallel/pvm_glb.shar
```

To unpack the library, type

```
sh pvm_glb.shar
```

Then, execute the script *make_glb*. This script will make the global library

```
~/pvm/pvm_glb/libpvm_glb.a
```

Also, the file

contains instructions on compiling and using the global library.

8.4 Observations on Using PVM

PVM is a low level communication tool and provides a few basic routines (about 40 routines). The PVM system does not come with global routines, so a library for doing global operations has been written.

PVM code is portable across architectures, but each node program needs to be compiled separately (unless the machines are NFS mounted). Preliminary results show that PVM performs very well for large grain parallel problems. For programs with intensive communication, PVM appears to have a large overhead due to communicating through LaRCNET.

Currently, there is no debugger. However, ORNL is developing one that is expected to be made available at the end of 1993. The debugger is XAB (X window Analysis and deBugger for PVM).

Another X interface to PVM is HeNCE (Heterogeneous Network Computing Environment). HeNCE allows the user to create a graphical representation of their parallel program. From the information in the graph, HeNCE creates all of the necessary communication using PVM. The next version of HeNCE was released at the end of 1992. The new release will contain a debugger and ParaGraph style traces.

The most difficult concepts of PVM to the user are installation and use on the iPSC/860. A C shell script has been written to automatically install the PVM software on any of the mentioned architectures. When using PVM on the iPSC/860, the host program has two roles: an iPSC/860 host and a PVM node. Since the PVM daemon is installed on the SRM, the PVM libraries are not used for inter processor communication on the Hypercube.

8.5 Known Bugs of PVM

The call **fbarrier** (**barrier** in C) does not allow the reuse of the barrier name variable. If another call is made to **fbarrier**, a different value for the barrier name must be used.

When using the message-passing routines, an integer must be used for passing the number of arguments. For example, when making a call to **fgetnfloat(FP,CNT,INFO)**, **CNT** must be an integer. If CNT is not an integer then the program may abnormally exit.

8.6 References for PVM

[1] iPSC/2 and iPSC/860 User's Guide

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 1993

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Parallel Software Tools at Langley Research Center

5. FUNDING NUMBERS
505-90-53-02

6. AUTHOR(S)
Stuti Moitra, Geoffrey M. Tennille, Christopher D. Lakeotes, Donald P. Randall, Jarvis J. Arthur, Dana P. Hammond and Gerald H. Mall

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-108995

11. SUPPLEMENTARY NOTES
Moitra, Tennille, Randall: NASA Langley Research Center, Hampton, VA
Lakeotes, Arthur, Hammond, Mall: Computer Sciences Corporation, Hampton, VA

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 60

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document gives a brief overview of parallel software tools available on the Intel iPSC/860 parallel computer at Langley Research Center. It is intended to provide a source of information that is somewhat more concise than vendor-supplied material on the purpose and use of various tools. Each of the chapters on tools is organized in a similar manner covering an overview of the functionality, access information, how to effectively use the tool, observations about the tool and how it compares to similar software, known problems or shortfalls with the software, and reference documentation.

It is primarily intended for users of the iPSC/860 at Langley Research Center and is appropriate for both the experienced and novice user.

14. SUBJECT TERMS
iPSC/860, performance monitor, debugger, heterogeneous computing environments, message passing.

15. NUMBER OF PAGES
89

16. PRICE CODE
A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102